Verwendung von Temporale-Differenz-Methoden im Schachmotor *FUSc#*

Diplomarbeit
Informatik/Mathematik-Fachbereich
Freie Universität Berlin

Autor: ***Marco Block***
Betreuer: ***Prof. Dr. Raúl Rojas***

August 2004

Zusammenfassung

In der Schachprogrammierung hat die Nutzung von Temporalen-Differenz-Methoden zur Optimierung der Bewertungsfaktoren einige Anläufe benötigt. Durch den Erfolg von Gerald Tesauros Backgammonprogramm *TD-Gammon* motiviert, welches den Weltmeister im Backgammon regelmässig schlägt, haben Jonathan Baxter, Andrew Tridgell und Lex Weaver mit ihrem Schachprogramm *KnightCap* die ersten sehr vielversprechenden Erfolge aufzeigen können. So änderten sie den von Richard Sutton veröffentlichten TD(λ)-Algorithmus zu dem TD-Leaf*(λ)*-Algorithmus, der die Koeffizienten der Bewertungsfunktionen eines Schachprogramms optimiert, indem nicht wie bei *TD-Gammon* mit den Wurzelknoten der Suchbäume, sondern mit den best forcierten Blattknoten gerechnet wird. Um dieses Vorgehen noch näher zu untersuchen und mögliche Ressourcen aufzuzeigen, wurde TD-Leaf(λ) in den Schachmotor *FUSc#* implementiert und getestet, welcher Zusammenhang zwischen der Evaluationsparameteranzahl und der Spielqualität besteht und eventuell so zu besseren Erfolgen führt. Dazu musste der Algorithmus zu TD-Leaf-ComplexEval(λ) erweitert werden. *FUSc#* kann die Koeffizienten der verschiedenen Stellungstypen individuell anpassen und optimieren. Eine Verbesserung der Spielstärke ist schon nach einigen Partien erkennbar, doch benötigt *FUSc#* für den Umfang der insgesamt über 56000 Faktoren in den 33 Stellungstypen bei einem gleichverteiltem Auftreten dieser, schätzungsweise weit über 50000 Trainingspartien. Dieser Testlauf wird leider erst nach Abgabe dieser Diplomarbeit beendet werden können.

http://www.inf.fu-berlin.de/~fusch

Inhaltsverzeichnis

Vorwort

Das Schachspiel beschäftigt den menschlichen Geist seit ca.1500 Jahren und stammt, nach letzten Erkenntnissen, aus Indien [Awerbach]. Möglicherweise ist das Schachspiel sogar noch viel älter, denn Galina Pugachenkova fand 1972 in Usbekistan zwei Spielfiguren, die auf das erste Jahrhundert nach Christus datiert wurden [First]. Damals fand es als Kriegsspiel Verbreitung, heute ist es ein Spiel zum Ausdruck von geistiger Kraft, tiefsinnigem Vorstellungsvermögen und konkreten Vorausberechnungen geworden. Die Komplexität des Schachspiels macht es für die Künstliche Intelligenz als Studienobjekt sehr interessant, man könnte sogar sagen: „Das Schachspiel ist die *drosophila melanogaster* der KI“. Auch in anderen Wissenschaften wird es als Beispiel für Komplexität gewählt.

> „Brian, wo liegt das wirkliche Problem der Wirtschaftswissenschaftler?“
>
> „Schach!“ antwortete Arthur, ohne nachzudenken.
>
> John H. Holland, Universität Michigan zu William Brian Arthur, Professor an der Universität Stanford nach dem Vortrag: „*Die Weltwirtschaft als adaptiver Prozeß*“

Nachdem der derzeit beste Schachspieler der Welt Garry Kasparov (Elo 2817) 1997 in einem Match über 6 Partien einem Schachcomputer (*Deep Blue*) unterlag, war die Welt aufgeschreckt.

Abbildung 1: Garry Kasparov

Im KaDeWe signierte mir Garry Kasparov sein neustes Buch (März 2004). Tausende Besucher wollten es sich nicht entgehen lassen, den besten Schachspieler aller Zeiten live zu erleben, der den unglaublichen Weltrekord mit einer ELO-Leistung von 2851 Punkten (Juli 1999) hält.

Garry Kasparov selbst sagte zuvor: „*Wenn ein Computer den Weltmeister schlagen kann, dann kann er die besten Bücher der Welt lesen, die besten Stücke schreiben, und alles über Geschichte, Literatur und Menschen kennen.*“ [King]. Dem war und ist nicht so - und es scheint eine Fiktion zu sein, den menschlichen Geist in silico anzufertigen. Trotz dieses verloren gegangenen Matches ist die Schachwelt davon überzeugt, dass Schachprogramme nicht wie Menschen spielen. Es fehlt die Tiefsinnigkeit, die Phantasie zweier Schachgroßmeister, die ein Schachspiel aufführen können, wie ein Theaterstück. Sicherlich ist nicht jedes Spiel ein Meisterwerk, aber es fällt dem erfahrenen Schachspieler nicht schwer den Unterschied zu einer Computerpartie herauszufinden. Der grosse Vorteil der Schachprogramme ist aber die zunehmende Rechenkraft, die es den Programmen ermöglicht, teilweise einige Millionen Stellungen pro Sekunde zu berechnen. Die von Jahr zu Jahr grösser werdenden Partiedatenbanken geben den Schachprogrammen auch einen sehr großen Vorteil, dem ein menschliches Gehirn wenig entgegensetzen kann.

Ein Schachprogramm besitzt eine Funktion, mit der es Schachstellungen bewerten kann. Der Umfang dieser Funktion kann von Programm zu Programm sehr unterschiedlich sein. Diese Funktion identifiziert verschiedene Stellungsmerkmale, bewertet diese und liefert die Summe der Bewertungen. Das Thema dieser Arbeit ist es, einen Schachmotor, die optimale Gewichtung der einzelnen Stellungsmerkmale in der Bewertungsfunktion durch aktives Spielen, eigenständig finden zu lassen. Dazu wurde der Schachmotor *FUSc#*

modifiziert und eine Temporale-Differenz-Methode verwendet. Einen Teil der theoretischen Vorarbeit habe ich in meiner Studienarbeit [Block] erörtert. Die Implementierung und eine mögliche Automatisierung des Verfahrens war ein Ziel dieser Diplomarbeit. Ich habe mich mit der Kernfrage beschäftigt, ob sich das Verfahren auch für eine sehr grosse Bewertungsfunktion eignet und bessere Ergebnisse liefern kann. Dazu wurde FUSc# mit einem Stellungsklassifikator ausgestattet und die Bewertungsfunktion auf 33 Stellungstypen erweitert.

Um in die Thematik einsteigen zu können, muss die Problematik der Koeffizientenjustierung bei Schachprogrammen verstanden sein, dazu gibt **KAPITEL 1** eine Übersicht zu den üblichen Bauplänen von Schachprogrammen und zeigt aktuelle Techniken und Methoden moderner Schachprogramme auf. Am Ende des Kapitels wird der Unterschied zwischen dem Schachverständis von Programmen und Schachmeistern diskutiert.

KAPITEL 2 widmet sich dem Reinforcement Learning, speziell der *Temporalen Differenz* (TD). Es hat einige Anläufe gebraucht, bis sich dieses Verfahren bei Strategie-Spielen wie Backgammon und Schach durchgesetzt hat. Den wohl größten Verdienst im Einsatz von TD in Schachprogrammen hatten Jonathan Baxter, Andrew Tridgell und Lex Weaver mit Ihrem Programm *KnightCap*. Dieses Programm wird genauer analysiert, da es den Basisalgorithmus darstellt, der im Schachmotor *FUSc#* verwendet wird.

Anschließend wird in **KAPITEL 3** der *FUSc#*-Stellungsklassifikator vorgestellt. Mit ihm soll eine umfassendere Stellungsbeurteilung möglich werden. Die Idee dahinter beruht auf der Tatsache, dass verschiedene Schachstellungstypen unterschiedliche Bewertungsparameter erfordern.

Das **KAPITEL 4** behandelt die Frage, ob der Ansatz mit TD-Methoden eine Justierung der Evaluationsparameter vorzunehmen, angewandt auf eine sehr viel größere Bewertungsfunktion (z.B. durch den Stellungsklassifikator) einen grösseren Erfolg liefern kann, und wie es möglich ist diese zu verwalten.

Die erhaltenen Ergebnisse und durchgeführten Experimente, werden im **KAPITEL 5** ausgewertet und diskutiert.

Kapitel 6 beinhaltet eine Anleitung zur Implementierung des vorgestellten Lernverfahrens, einen Verweis auf die aktuellen Schachregeln der FIDE, die Tic-Tac-Toe-Regeln und Informationen über den zur Leistungsmessung verwendeten Schachserver.

Vielen Dank an *Prof. Dr. Raúl Rojas*, der meine Diplomarbeit betreut und mir dieses Thema angeboten hat. Mit Vorträgen und Präsentationen zum Thema fördert er die wissenschaftliche Herangehensweise und in Diskussionen können Probleme angegangen und Fehler schneller gefunden werden.

Mit Begeisterung widme ich mich dem Schachspiel nun schon viele Jahre. Begonnen hatte alles mit meinem Großvater, der mir das Schachspiel im Alter von 6 Jahren beibrachte, viel mit mir spielte und mich schon in frühen Jahren vor den Computer setzte und damit meine Informatikkarriere mitverschuldete. Ich möchte mich für die Unterstützung, die in mich investierte Zeit und die Liebe bedanken, die meine Großeltern all die Jahre für mich aufbrachten und dies noch immer tun. Bedanken möchte ich mich bei meinen Eltern, die es immer versucht und geschafft haben, mir ein sorgenloses Studium zu ermöglichen - meiner Freundin Katrin, die immer zu mir hält und mir die Kraft gibt in die Zukunft zu schauen.

Meine Diplomarbeit hätte nicht geschrieben werden können, wenn das *FUSc#*-Team nicht gewesen wäre. Viel Unterstützung habe ich von Andre Rauschenbach erfahren. Auch ein Dankeschön an Miguel Domingo, der mir ein guter Ratgeber, objektiver Kritiker aber ein noch besserer Freund ist.

Hiermit bestätige ich, dass ich diese Diplomarbeit selbstständig angefertigt und alle verwendeten Textpassagen, Formeln und Zitate als solche markiert und im Quellenverzeichnis angegeben habe.

Marco Block

Kapitel 1

Einführung

1.1 Motivation

Der Komplexitätsgrad des Schachspiels macht es Rechnern heute nicht möglich, dass Schachspiel endgültig zu erfassen und komplett durchzurechnen. Es wird sicherlich noch viele Computergenerationen dauern, bis ein Schachprogramm zu jedem beliebigen Zug aus der Startstellung folgende Antwort liefern könnte: „...erzwungenes Remis in 114 Zügen.“. Ein menschlicher Schachspieler, der auf hohem Niveau spielt, benötigt keine so tiefe Vorausberechnung. Er besitzt Intuition und eine über die Jahre erlernte, selektive Zugwahl und Stellungsbewertung. Schachmeister rechnen laut [Zipproth] im Schnitt 1.2 Züge pro Stellung, da aber Schachprogramme[1] über keine Intuition verfügen, müssen sie alle Züge in einer Stellung in Betracht ziehen, um eine Bewertung vornehmen zu können. Anschließend geben sie den Zug zurück, mit dem vielversprechensten Weg. Stefan Zipproth gibt an, dass ein Schachprogramm, um Großmeisterniveau zu erreichen, etwa 16 Halbzüge[2] in längeren Partien[3] vorrausrechnen müsste. Angenommen, es gibt im Schnitt 20 Züge pro Stellung und ein Programm würde ohne Optimierungstechniken die Halbzugtiefe von 16 erreichen, dann müßten ca.

$$600.000.000.000.000.000.000$$

mögliche Varianten betrachtet werden [Zipproth].

Die Konzentration bei der Entwicklung von Schachprogrammen liegt demzufolge auf der Brute-Force-Strategie. Es werden viele Pruning-Verfahren[4] und eine lineare Stellungsbewertungsfunktion, die einige schachtheoretische Kenntnisse auf konkrete Stellungen anwendet um eine Bewertung vorzunehmen, verwendet. Das grösste Problem ist die Gewichtung der einzelnen in dieser Funktion auftretenden Faktoren. Das in dieser Diplomarbeit untersuchte TD(λ)-Verfahren hilft, dieses Problem vom Schachmotor eigenständig lösen zu lassen.

1.2 Schachprogrammierung eine Übersicht

1.2.1 Grundbauplan

Der Schachmotorentwurf beginnt mit der internen Schachbrettrepräsentation. Diese sollte möglichst effizient in der Lage sein, eine regelkonforme Zugliste für eine konkrete Stellung zu liefern. Ziel ist es einen

[1] Damit sind die stärksten aktuellen Schachprogramme gemeint. Es gibt zwar Ansätze, die „Intuition im Schachspiel“ für Schachprogramme mit Neuronalen Netzen zu lösen, aber bisher blieben große Erfolge aus.

[2] Jedes Bewegen einer Figur wird im Schach *Halbzug* genannt. Nachdem einmal Weiß und einmal Schwarz gezogen haben, ist demzufolge ein *Zug* gespielt. Wenn beispielsweise Weiss im 8. Zug Matt setzen kann, wurden im Spiel $7*2+1=15$ Halbzüge gespielt.

[3] Beispielsweise der Modus: 2 Stunden für die ersten 40 Züge und 30 Minuten für die restlichen Züge.

[4] Um nicht den kompletten Variantenbaum berechnen zu müssen, gibt es zahlreiche Optimierungsverfahren, die Zweige des Suchbaumes kürzen und somit den Aufwand bei gleichbleibender Spielqualität minimieren.

Schachmotor zu entwickeln, der bei der Eingabe einer Schachstellung in gegebener Zeit den besten Zug (manchmal reicht auch ein guter aus) zurückzugeben. Um das zu erreichen, ist zum einen viel Schachwissen (siehe 1.2.1.3) notwendig und zum anderen ein Algorithmus (siehe 1.2.1.2), der erstmal intern einen Zug, oder eine ganze Zugfolge ausprobiert und die erhaltene Stellung nach seinen Kriterien bewertet. Die Bewertung liefert einen Zahlenwert, wobei ein positiver Wert einen weissen Vorteil und ein negativer Wert einen schwarzen Vorteil angibt. Die Ausstattung eines Schachmotors kann noch beliebig erweitert werden (z.B. Eröffnungsbuch, grafische Benutzeroberfläche).

1.2.1.1 Brettdarstellung und Zuggenerierung

Begonnen wird jeder Schachmotor mit der Darstellung der Figuren auf dem Brett. Jedes Modell besitzt Vor- und Nachteile bezüglich der Zuggenerierung. Im folgenden werden verschiedene Darstellungen diskutiert.

8×8-**Matrix**

Eine der ersten und einfachsten Versuche für die Darstellung eines Brettes in einem Programm ist die 8×8-Matrix. Dazu wird ein zweidimensionales Array mit folgenden Werten gefüllt: 0 keine Figur, 1 Bauer, 2 Springer, 3 Läufer, 4 Turm, 5 Dame, 6 König. Für die Farbunterscheidung werden die weißen Figuren mit positiven und die schwarzen mit negativen Zahlen angegeben. Daraus ergibt sich folgende Matrix für die Startstellung:

-4	**-2**	**-3**	**-5**	**-6**	**-3**	**-2**	**-4**
-1	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**
4	**2**	**3**	**5**	**6**	**3**	**2**	**4**

Um nun legale Züge zu einer Stellung zu generieren, wird in einer Schleife die Matrix durchgelaufen und bei entsprechend identifizierter Figur die erlaubten Zielfelder betrachtet. Steht auf dem Feld eine gegnerische Figur, wurde beispielsweise ein Schlagzug ermittelt. Sollte sich auf dem Zielfeld eine eigene Figur befinden, oder das Zielfeld außerhalb der Matrix liegen, was einige zusätzliche Abfragen benötigt, so wird an dieser Stelle abgebrochen und das nächste Matrixfeld untersucht. Ist ein Zug regelkonform, so wird er an die Zugliste gehängt.

10×12-**Matrix**

Die 10×12-Matrix-Darstellung hat gegenüber der 8×8-Matrix einen kleinen Vorteil. Dazu zunächst die Startstellung:

99	99	99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99	99	99
99	**-4**	**-2**	**-3**	**-5**	**-6**	**-3**	**-2**	**-4**	99
99	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**	**- 1**	**-1**	99
99	0	0	0	0	0	0	0	0	99
99	0	0	0	0	0	0	0	0	99
99	0	0	0	0	0	0	0	0	99
99	0	0	0	0	0	0	0	0	99
99	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	99
99	**4**	**2**	**3**	**5**	**6**	**3**	**2**	**4**	99
99	99	99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99	99	99

Die Matrix wurde um einen Rand erweitert. Ein ungültiger Zug, der die Matrixränder überschreitet kann schneller identifiziert werden. Ein Zielfeld gilt nun als „legal“, wenn auf ihm keine eigene Figur steht und der Wert kleiner 99 ist. Damit lässt sich die Performance etwas verbessern.

BitBoards und Rotated BitBoards

Die Verwendung der BitBoardtechnologie hat sich in der Schachprogrammierung, neben dem Einsatz von Alpha-Beta-Algorithmen, als größte Innovation herausgestellt. Heutzutage setzen alle professionellen Schachprogrammierer auf diese Repräsentation, da sie doch die Vorzüge der bitbasierten Datenverarbeitung in einem Prozessor ausnutzt. So werden die Figurentypen (egal welcher Spielfarbe) in unterschiedlichen 64-Bitworten gespeichert. Ein Bit steht für ein Feld und gibt an, ob an dieser Stelle diese Figur steht oder nicht. Zusätzlich wird je ein Bitwort für die weissen und die schwarzen Figuren (`white_occupied`, `black_occupied`) gespeichert, so dass mit folgender, einfachen Operation die weissen Springer zu identifizieren sind:

```
white_knights  = board.knights & board.white_occupied;
```

Die Zuggenerierung erfolgt auch sehr einfach, beispielsweise können die Bauernzüge (ein Feld vor) durch eine einfache Shift-Operation generiert werden:

```
// liefert ein Bitwort mit den Zielfeldern der weissen Bauern
white_pawn_tos = (board.white_occupied & board.pawns) >> 8
```

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**

(&

0	0	0	0	0	0	0	0
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**
0	0	0	0	0	0	0	0

) >> 8 =

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Um nun individuelle Figurenmerkmale zu bewerten, stellt dich die Frage, wie es schnell möglich ist, eine einzelne Figur auf dem Brett zu finden (z.B. ein Springer) und diesen dann separat zu bearbeiten. Dazu gibt es zwei Funktionen[5], die eine kann in einem Bitwort zunächst die letzte auftretende 1 identifizieren und so ein Bitwort bestehend aus Nullen und dieser 1 liefern und eine weitere, die diese 1 dann löscht. So spart man sich das elementeweise Durchlaufen in einer Schleife.

bitwort & -bitwort ...liefert das Bitwort, mit der letzten gesetzten 1

Beispiel: Sei nun b=01010 mit dem Ziel, ein Bitwort zu erhalten, dass nur die letzte in b gesetzte 1 enthält. Um in der Zweierkomplementdarstellung ein Bitwort zu negieren, werden zunächst die einzelnen Bits gekippt und anschliessend zum Bitwort eine 1 addiert:

$$\begin{aligned} 01010 \,\&\, -01010 &= 01010 \,\&\, (10101 + 1) \\ &= 01010 \,\&\, 10110 \\ &= 00010 \end{aligned}$$

bitwort = bitwort & (bitwort - 1) ...löscht die letzte 1 im Bitwort.

[5] Ist möglich, da die Darstellung im Zweierkomplement vorliegt. Bei der Negation eines Bitwortes, werden alle Einträge invertiert und das erhaltene Bitwort anschliessend um 1 aufsummiert.

Als nächstes soll diese letzte 1 aus dem Bitwort entfernt werden, dazu:

$$\begin{aligned} 01010 \,\&\, (01010 - 1) &= 01010 \,\&\, 01001 \\ &= 01000 \end{aligned}$$

In einer Schleife können nun alle gesetzten Bits in einem Bitwort durchlaufen werden:

```
b = board.knights & board.white_occupied;
// solange Springer auf dem Brett vorhanden sind
while (b = b&-b)
{
   /* in b steht nun ein Springer und kann entsprechend
      verarbeitet werden... */

   b &= b-1;   // entferne diesen Springer
}
```

Der Phantasie sind nun keine Grenzen gesetzt. Verschiedenste Muster lassen sich in einem Bitmuster repräsentieren und entsprechend effizient verarbeiten.

Die *Rotated Bitboards* verwenden einen weiteren Trick für die Leistungssteigerung. Um beispielsweise die Läufer- und Turmzüge schneller zu generieren, existiert das BrettBitwort (gesamtes Brett), auf dem alle Figuren (schwarz und weiss) stehen, in vier Varianten:

```
private int[] normal = {
         A1,B1,C1,D1,E1,F1,G1,H1,
         A2,B2,C2,D2,E2,F2,G2,H2,
         A3,B3,C2,D3,E3,F3,G3,H3,
         A4,B4,C3,D4,E4,F4,G4,H4,
         A5,B5,C4,D5,E5,F5,G5,H5,
         A6,B6,C5,D6,E6,F6,G6,H6,
         A7,B7,C6,D7,E7,F7,G7,H7,
         A8,B8,C7,D8,E8,F8,G8,H8
};
```

```
private int[] normal_to_l90 = {
         A1,A2,A3,A4,A5,A6,A7,A8,
         B1,B2,B3,B4,B5,B6,B7,B8,
         C1,C2,C3,C4,C5,C6,C7,C8,
         D1,D2,D3,D4,D5,D6,D7,D8,
         E1,E2,E3,E4,E5,E6,E7,E8,
         F1,F2,F3,F4,F5,F6,F7,F8,
         G1,G2,G3,G4,G5,G6,G7,G8,
         H1,H2,H3,H4,H5,H6,H7,H8,
```

```
private int[] normal_to_a1h8 = {
         A1,B2,C3,D4,E5,F6,G7,H8,
         A2,B3,C4,D5,E6,F7,G8,H1,
         A3,B4,C5,D6,E7,F8,G1,H2,
         A4,B5,C6,D7,E8,F1,G2,H3,
         A5,B6,C7,D8,E1,F2,G3,H4,
         A6,B7,C8,D1,E2,F3,G4,H5,
         A7,B8,C1,D2,E3,F4,G5,H6,
         A8,B1,C2,D3,E4,F5,G6,H7
};
```

```
private int[] normal_to_a8h1 = {
         A1,B8,C7,D6,E5,F4,G3,H2,
         A2,B1,C8,D7,E6,F5,G4,H3,
         A3,B2,C1,D8,E7,F6,G5,H4,
         A4,B3,C2,D1,E8,F7,G6,H5,
         A5,B4,C3,D2,E1,F8,G7,H6,
         A6,B5,C4,D3,E2,F1,G8,H7,
         A7,B6,C5,D4,E3,F2,G1,H8,
         A8,B7,C6,D5,E4,F3,G2,H1
```

Das Bitwort *normal* repräsentiert die „normale“ Brettstellung aller Figuren, wobei beim Lesen darauf zu achten ist, dass der erste Eintrag links, unten auf dem Brett zu finden ist und die Zeilen dementsprechend von oben nach unten verlaufen. Im Bitwort *normal_to_l90* steht das Wort um 90° nach links gedreht. Die beiden Bitworte *normal_to_a1h8* und *normal_to_a8h1* liefern die um 45° gedrehte Brettdarstellung. Der Vorteil liegt nun bei der schnelleren Zuggenerierung. Bei der Initialisierung wurden alle möglichen

Reihen- und Spaltenkonfigurationen ermittelt und die entsprechend möglichen Züge von jedem Feld aus berechnet und in einer Datenstruktur gespeichert. In der Suche müssen diese nun nicht generiert, sondern einfach ausgelesen werden. Rotated Bitboards sind nun desshalb nötig, um z.B. bei den Läufern die vorab berechneten Werte aus der Datenstruktur verwenden zu können, da die Bits der Diagonalen in der 45° gedrehten Brettdarstellung direkt aufeinanderfolgen. Sie können nun zusätzlich zum Startfeld als Index in die Datenstruktur genommen werden, um von einem Feld (Startfeld) alle möglichen Züge bei gegebener Diagonalkonfiguration zu erhalten. Analoges gilt für Türme und Damen. Ein Nachteil ist allerdings, dass alle vier BitWorte bei Ausführung und Zurücknahme eines Zuges aktualisiert werden müssen.

Auf den neuen 64-Bit-Prozessoren wird sich die Performance dieser Brettdarstellung erheblich verbessern.

ToPiecesArray

Diese Brettrepräsentation ist keine sehr verbreitete. Sie wird hier aufgeführt, da der später besprochene KnightCap-Schachmotor diese Brettdarstellung verwendet. Im Gegensatz zu der BitBoarddarstellung wird zusätzlich zu jedem Feld ein 32-BitWort gespeichert. Die 32-Bit stehen für die 32 Startfiguren[6]. Betrachtet man nun das Feld j aus dem 64-Bit-Array, dann steht das i-te Bit für eine Figur i, die das Feld j attackiert. Aus Abbildung 1.1 ist zu entnehmen, dass nur die 32-te Figur auf das Feld j wirkt[7], beispielsweise könnte das der schwarze Turm sein. Durch diese Modellierung lassen sich komplizierte Faktoren der Stellungsbewertung, die sehr oft große Rechenzeit in Anspruch nehmen, leicht beschreiben. Es werden 64×32 Bit für das Brett gespeichert. Die Brettfelder beginnen mit dem Feld $a1$ und sind zeilenweise, aufsteigend durchnummeriert. Das Problem dabei besteht in der Aktualisierung der Bitworte, nachdem ein Zug ausgeführt bzw. zurückgenommen wurde. Dieser Aufwand ist recht hoch. Positiv hingegen ist beispielsweise die Möglichkeit, schnell an wichtige Daten heranzukommen. Dabei kann z.B. die Frage schnell beantwortet werden, ob ein König im Schach steht[8].

1.2.1.2 Zugwahlalgorithmen und Optimierungstechniken

MinMax (Brute-Force-Strategie)

Zum Schachspiel gehören zwei Spieler. Es wird abwechselnd gezogen. Angenommen, Schach wäre ein sehr berechenbares und überschaubares Spiel wie z.B. Tic-Tac-Toe, dann könnte der Computer alle möglichen Zugfolgen in einem Baum berechnen und die Endstellungen mit SIEG, REMIS, NIEDERLAGE bewerten (aus Sicht des am Zug befindlichen Spielers). Es wird davon ausgegangen, dass jeder Spieler den für sich besten Zug spielt. Begonnen wird in der untersten Baumebene. Die Bewertung der jeweils beste Spielmöglichkeit wird an den Vaterknoten weitergeleitet. Zuletzt erhält die Wurzel einen Wert und kann nun entsprechend den ersten Zug ausführen mit der Gewissheit, dass er mindestens das forcierte Ergebnis erreichen kann. Möglicherweise macht der Gegner einen Fehler, aber darauf wird nicht spekuliert. Das für die Künstliche Intelligenz interessante am Schachspiel ist die Komplexität. Eine Zugfolge bis zum Spielende berechnen zu können ist derzeit nicht möglich. Es reicht an bestimmten Stellen sagen zu können, hier ist das materielle Verhältnis (wahrscheinlich) nicht ausreichend, um gewinnen zu können. In diesem Fall kann eine schlechte Bewertung an den Vaterknoten übergeben werden.

Hilfe leistet folgende Vereinbarung: Für den weißen Spieler werden Stellungsvorteile positiv und für den Spieler mit den schwarzen Steinen negativ bewertet. Demzufolge ist Weiß am Zug bestrebt, einen möglichst großen Wert zu erhalten und wird, an welcher Stelle am Baum er auch am Zug ist, den maximalen wählen. Der Gegner wählt das Minimum usw. Da nun eine Variantenvorrausberechnung im Schach nicht unendlich tief gehen kann und dies mit der aktuellen Technik auch nicht möglich ist, muss die Berechnung in einer bestimmten Tiefe abgebrochen und eine Bewertung der entstandenen Stellung vorgenommen werden. Abbildung 1.2 gibt ein Beispiel für die Vorgehensweise dieses Algorithmus.

Algorithmus 1 zeigt den MinMax-Algorithmus im PseudoCode.

[6] Es können keine Figuren hinzukommen, lediglich wegfallen oder sich umwandeln.

[7] Mit Wirkung einer Figur auf ein Feld ist gemeint, dass die Figur im nächsten Zug dorthinziehen könnte oder dem gegnerischen König den Zugang auf dieses Feld verwehrt.

[8] Dieses Problem wird in der Schachprogrammierung viel diskutiert [Stein]. Ein Zug kann nur als legal angesehen werden, wenn er das „im Schach“ stehen des eigenen Königs verhindert und nicht nach sich zieht.

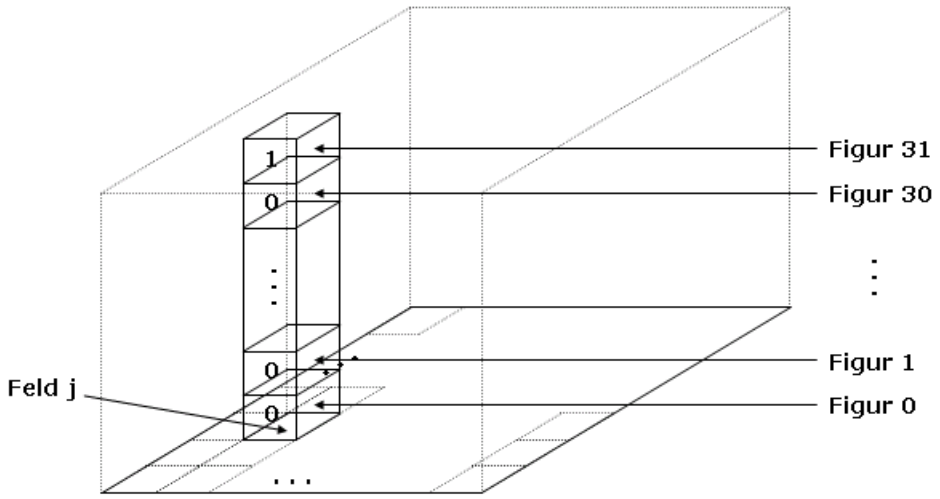

Abbildung 1.1: ToPiecesArray

Für jedes Feld auf dem Schachbrett wird zusätzlich ein 32-BitWort gespeichert. Darin enthalten sind die möglichen Zugfelder der Figuren. Auf der Abbildung zu sehen, besitzt die Figur mit dem Index 31, welche z.B. den schwarzen Turm repräsentieren könnte, die Möglichkeit auf dieses Feld zu ziehen. Andere Figuren haben, wegen der 0en in „ihrem“ Index, nicht die Möglichkeit auf dieses Feld zu ziehen.

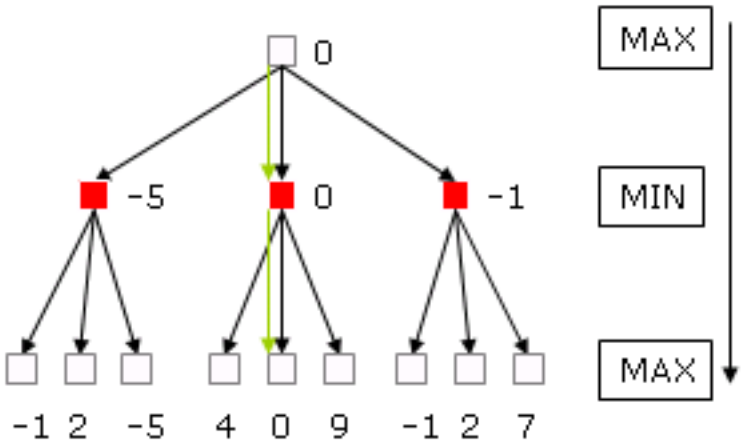

Abbildung 1.2: MinMax-Algorithmus.

In diesem Beispiel wird der komplette Suchbaum bis zur Tiefe 2 betrachtet. Die Stellungen in den Blattknoten werden durch eine Bewertungsfunktion evaluiert. Da davon ausgegangen wird, dass jeder Spieler den für sich besten Zug wählt und ein positiver einen weißen Vorteil und ein negativer Wert einen schwarzen Vorteil angibt, wird im linken Teilbaum der Wert -5 nach oben gegeben. Da schwarz am Zug ist, wird er den für sich besten wählen. Bei den Knoten der Tiefe 1 ist schwarz jeweils am Zug und wählt den minimalen Knoten. Von der Wurzel aus ist Weiß am Zug und wählt das Maximum, also 0.

Algorithm 1 MinMax-Algorithmus

```
// Eingabe: Tiefe t, maximale Suchtiefe und Farbe die am Zug ist
// wenn wtm=true, ist Weiß am Zug, ansonsten Schwarz
// Die Funktion liefert die bestmögliche Bewertung von dieser
// Stellung aus.
int minmax(t, wtm) {
   if (t==0)
      if (wtm) return  evaluiereStellung();
      else     return -evaluiereStellung();
   generiereZüge();
   bestwert = 0;
   while (Züge vorhanden){
      mache(nächsten_Zug);
      wert = -minmax(t-1, -wtm);
      nimmZugzurück();
      if (wert>bestwert)
         bestwert = wert;
   }
   return bestwert;
}
```

AlphaBeta

Der MinMax-Suchbaum wird nach wenigen Zügen sehr groß. Eine Idee ist es, lokal obere und untere Schranken für die Suche festzulegen. Diese Schranken sollen zu jedem Zeitpunkt im Baum Auskunft geben können, was die am Zug befindliche Partei und dazu der Gegner bereits forcieren können. Dann ließe sich in dem Fall, dass eine Stellung erreicht wird, die besser für unseren Gegner ist, als eine bereits forcierte, darauf verzichten, die restlichen Züge in dieser Ebene weiter zu betrachten. Das Abschneiden der weiteren Züge im Baum nennt man CutOff (oder Beta-CutOff). Ein kleines Beispiel dazu ist der Abbildung 1.3 zu entnehmen.

Die Reihenfolge der Zuglisten spielt dabei eine entscheidende Rolle. Im schlechtesten Fall wird wie bei dem vorher besprochenen MinMax-Algorithmus der komplette Suchbaum bis zu einer bestimmten Tiefe berechnet (brute-force) und demnach alle n möglichen Positionen evaluiert. Im besten Fall stünden die besten Züge immer vorn und das hiesse, nur $\sqrt{n}$ Positionen müssten evaluiert werden [Heinz].

Ruhesuche

Im Schachspiel ist es problematisch einem Schachcomputer nur eine bestimmte Suchtiefe zu erlauben. Beispielsweise könnte von der Startstellung aus, Weiß nach 3 Zügen einen Bauern gewinnen. Das Programm sieht dann noch nicht, dass die Figur, die den Bauern geschlagen hat, im nächsten Zug ebenfalls vom Brett genommen werden kann. Eine Bewertung hätte aber ergeben, dass Weiß in dieser Stellung materiell besser steht und demzufolge den Bauern schlagen sollte. Um dieses Problem zu umgehen, wird die sogenannte Ruhesuche nach einer vorgegebenen Tiefe ausgeführt. Anstatt alle Züge im weiteren zu betrachten, werden nur Schach- und Schlagzüge ab einer bestimmten Tiefe weiter untersucht. Das verhindert genau das geschilderte Problem, tut aber ein neues auf. Eine Stellung die sehr viele Schlagzüge enthält, benötigt dementsprechend auch wesentlich mehr Zeit zur vollständigen Untersuchung.

Die Ruhesuche beinhaltet viele Optimierungsressourcen, da die Zugsortierung auch dort eine wesentliche Rolle spielt. Zu den vielversprechensten Sortierungen zählt die folgende: Sortierung zunächst nach der zu schlagenden Figur (Dame, Turm, ...), anschließend wird als zweites Kriterium die schlagende Figur sortiert (Bauer, Springer, ...). Demnach versuchen wir mit der „schwächsten“ Figur die materialtechnisch gesehene „stärkste“ zu schlagen.

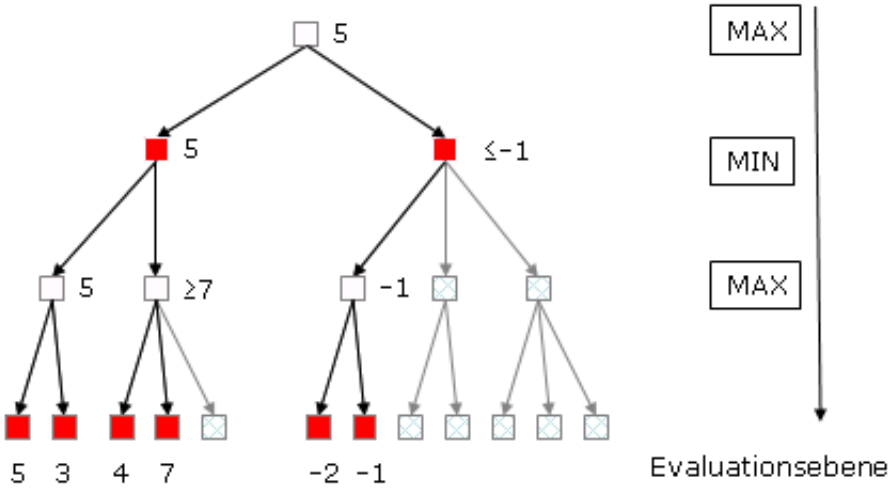

Abbildung 1.3: AlphaBeta-Algorithmus.

In diesem Beispiel wird der Baum wie in der Tiefensuche durchlaufen. Die ersten beiden Evaluationenswerte sind 5 und 3. Da von dieser Stellung aus Weiß am Zug ist, wird er das Maximum wählen, also 5. Nun wird der nächste Ast untersucht, dort ergibt die Bewertung eine 4. Damit hätte Schwarz bereits einen besseren Zug für sich gefunden, aber es könnte für Weiß noch einen besseren geben. Die folgende Evaluation liefert eine 7. Nun kann die weitere Suche abgebrochen werden, denn Schwarz wird diesen Zug nicht spielen, denn er kann bereits eine für sich bessere Variante forcieren, die den Wert 5 liefert. Nach diesem AlphaBeta-Verfahren erhalten wir am Ende den Wert 5 für die Wurzel und sehen, dass einige Evaluationen und Zugberechnungen ausgelassen werden konnten.

Hauptvarianten & iterative Suche

Zum Nachvollziehen der „Ideen“ des Schachmotors ist es ratsam die Hauptvarianten[9] mitzuspeichern. Das sind die forcierten Zugfolgen, die sich bei bestmöglichem Spiel der beiden Spieler ergeben. Zum Beispiel:

```
e2e4 e7e5 g1f3 b8c6 f1b5 a7a6
```

In Abbildung 1.2 repräsentiert die Hauptvariante den Pfad der Knoten von der Wurzel bis zum Blattknoten mit dem Wert 0 und in Abbildung 1.3 den Pfad von der Wurzel zum Blattknoten mit dem Wert 5. Die Hauptvariante wird nun verwendet, um eine iterative Suchtiefenberechnung zu ermöglichen. Zunächst wird die Tiefe t berechnet und die dabei ermittelte Hauptvariante gespeichert, anschliessend lässt sich die Suchtiefe $t+1$ schneller untersuchen. Der Grund dafür ist die Zugsortierung. Die Hauptvariantenzüge werden zuerst ausgeführt, falls diese in dieser Tiefe in der generierten Zugliste vorkommen. Es wird davon ausgegangen, dass sich die Hauptvarianten nicht so oft ändern und schon einen recht genauen Wert für die Suche liefern. Dementsprechend führt ein früh gefundener guter Wert zu vielen CutOffs im Alpha-Beta-Algorithmus. Treibt man diese Idee auf die Spitze, so wird einfach bei Tiefe 1 begonnen und solange in einer Schleife die Tiefe inkrementiert, bis der Zeitvorrat für die Zugsuche[10] aufgebraucht ist.

Anzumerken ist, dass bei der späteren Verwendung von Transpositionstabellen, auf die Hauptvarianten verzichtet werden kann, da die besten Züge bereits als solche dort gespeichert sind. Parallel wird die Hauptvariante trotzdem gespeichert, um nach Berechnung eines Zuges (aus Testzwecken) den nachvollziehen zu können, auf welcher Basis der Schachmotor den besten Zug gewählt hat und welchen Pfad er als besten ansieht. Aus Gründen der Fehlerfindung ist dies unersetzlich.

Zugsortierungen

Die Zugsortierung ist das wichtigste Kriterium für eine erfolgreiche Anwendung des Alpha-Beta-Algorithmus (viele CutOffs). Angenommen die Züge seien von ihrer Qualität andersherum sortiert, dann arbeitet der Alpha-Beta-Algorithmus auf dem Niveau von MinMax, da er keine CutOffs finden kann. Die Aufgabe der Schachprogrammierer ist es daher eine möglichst gute Sortierung zu finden. Die Hauptvariante stellt ein sehr gutes Kriterium dar, aber auch Heuristiken, wie z.B. die Killerheuristik (die Züge, die in ihrer Tiefe die besten waren bekommen einen Bonus und landen bei einer Zugsortierung weiter vorn) können eine bessere Zugsortierung liefern.

Nullmoves

Als vielversprechende Möglichkeit Äste des Suchbaums vor einer festgesetzten Tiefe t abzuschneiden, hat sich das Nullmovepruning etabliert. Die Idee dabei ist, dass die am Zug befindliche Partei als ersten Zug einfach nichts macht und somit der Gegner zweimal hintereinander am Zug ist. Das klingt ersteinmal etwas kurios, begründet sich aber dahingehend, dass eine Stellung die schlecht ist, als solche noch schneller zu finden ist. Wenn eine Partei zwei Züge zur Verfügung hat und keinen Vorteil daraus ziehen kann, so muss diese Stellung für diese Partei schlecht sein und die Suche kann beendet werden. Ausnahmen stellen Situationen dar, die mit Zugzwang oder Schachstellungen in Zusammenhang stehen.

Die Leistungssteigerung beläuft sich laut [Stein] bei ca. $10-20\%$, oder ca. $200-300$Elopunkte.

Transpositionstabellen

Bisher wurde von einem Suchbaum gesprochen, doch handelt es sich im Schach wirklich um einen Baum, oder eher um einen Graphen? Es können beispielsweise Stellungen durch verschiedene Zugfolgen erreicht

[9] Hauptvarianten werden z.B. als Mittel der Kommentierung von Schachpartien verwendet. Sollte ein Spieler beispielsweise eine taktische oder strategische Zugmöglichkeit in der Partie versäumt haben, wird diese als Alternative angegeben.

[10] Normalerweise steht dem Schachmotor zur Ermittlung des besten Zuges in einer Stellung eine bestimmte Zeit zur Verfügung. Diese ist relativ zur noch vorhandenen Gesamtzeit der Partie. Es gibt auch Strategien, eine bestimmte Suchtiefe zu berechnen und erst dann den besten Zug zu liefern, dass kann aber zum Ende der Partie zu großen Zeitproblemen führen.

werden. Nun würde ein weiteres Berechnen von diesen Stellungen aus, möglicherweise sehr viel unnötige Zeit in Anspruch nehmen. Transpositionstabellen sollen dieses Problem lösen. Abhängig von der gerade berechneten Tiefe eines Knotens, lassen sich 3 Fälle unterscheiden.

> **1. Fall (EXACT):** Alle Züge in der Zugliste sind abgearbeitet. Es gab einen besten Zug. Kein Beta-CutOff.
>
> *Konsequenz:* Sollte diese Stellung nocheinmal anzutreffen sein, so besitzt der berechnete Wert auch jetzt noch Gültigkeit. Demnach wird der Wert zurückgeliefert.

```
return HashWert(der aktuellen Stellung);
```

> **2. Fall (UPPER):** Alle Züge in der Zugliste sind abgearbeitet. Es gab keinen besten Zug. Kein Beta-CutOff.
>
> *Konsequenz:* Bei einer Wiederholung dieser Stellung, kann lediglich die obere Schranke aktualisiert werden. Sollte der Wert aber kleiner als der bisher forcierte alpha-Wert sein, wird er zurückgeliefert.

```
if (HashWert(der aktuellen Stellung)<=alpha)
   return HashWert(der aktuellen Stellung);
if (HashWert(der aktuellen Stellung)<beta)
   beta = HashWert(der aktuellen Stellung);
// Berechnung dieses Teilbaumes geht weiter
```

> **3. Fall (LOWER):** Es gab einen Beta-CutOff.
>
> *Konsequenz:* Nur die untere Schranke kann aktualisiert werden, sollte der Wert aber grösser als der aktuelle Beta-Wert sein (also der Wert, den der Gegner forcieren kann), so beenden wir die Suche und liefern den Wert zurück.

```
if (HashWert(der aktuellen Stellung) >= beta)
   return HashWert(der aktuellen Stellung);
if (HashWert(der aktuellen Stellung)>alpha)
   alpha = HashWert(der aktuellen Stellung);
// Berechnung dieses Teilbaumes geht weiter
```

Laut [Stein] steigert diese Technik die Leistung um ca. $5 - 10\%$ oder $50 - 100$ Elopunkte.

Suchfensterminimierung

Schachmeister gehen davon aus, dass sich eine Partie bei den „besten“ gespielten Zügen ständig im Gleichgewicht befindet. Wenn davon ausgegangen werden kann, dass der Schachmotor und sein Gegenüber keine Fehler machen und beide gute Züge spielen, so könnte die letzte berechnete Bewertung einer Stellung (mit anschließender Ausgabe eines Zuges) gespeichert werden und nach dem anschließenden gespielten Zug des Gegners benutzt werden, um mit dem Alpha-Beta-Verfahren schneller den besten Zug zu finden.

> **Vorher:** wert = AlphaBetaAlgorithmus(-∞, ∞, ...)

Mit einer gewissen Wahrscheinlichkeit kann sich der Wert nach oben oder unten verschieben, dazu öffnet man ein kleines Fenster um den alten Schätzwert `alterwert`. Beispielsweise könnte man ein Fenster der Breite 10 öffnen:

> fenster = 5
>
> **Nachher:** neuerwert = AlphaBetaAlgorithmus(alterwert-fenster, alterwert+fenster, ...)

Der MTD(f)-Algorithmus verfolgt diese Idee noch intensiver [Plaat].

Permanent Brain

Normalerweise berechnet ein Schachspieler mögliche Zugfolgen und bewertet Stellungen, auch wenn er nicht am Zug ist. Das gleiche lässt sich mit einem Schachprogramm realisieren. Ausgegangen wird von dem Zug des Gegners in der ermittelten Hauptvariante. Sollte dieser gespielt werden, wurde möglicherweise sehr viel Rechenzeit gespart. Falls nicht, sind aber die Transpositionstabellen mit den Folgestellungen gefüllt und ermöglichen so eine schnellere Abarbeitung.

1.2.1.3 Stellungsbewertung

Einen Schachmotor auf den technisch aktuellen Stand zu bringen, was Brettrepräsentation und Zugwahlalgorithmen betrifft, ist heutzutage aufgrund der Fülle an Informationen nicht mehr schwierig[11].

Worin unterscheiden sich die Schachmotoren dann eigentlich noch? Es ist die Bewertung einer Stellung. In ihr werden alle schachtheroretischen Aspekte behandelt. So kann ein Schachmotor, der sehr tief rechnet, aber nur auf das Material fixiert ist im Normalfall keinen Schachmeister schlagen. Es gehören all die Strukturen und Muster dazu, die ein Schachspieler im Laufe seines Schachstudiums ebenfalls erlernt, beispielsweise dass gewisse Bauernkonstellationen eher schwach und andere als stark zu beurteilen sind. Im folgenden sind exemplarisch einige Bewertungskriterien vorgestellt. Dabei sei angemerkt, dass wir immer aus der Sicht von Weiß bewerten, dass heißt wir liefern einen positiven Wert, in dem Fall, dass die weißen Vorteile die schwarzen überwiegen.

Material

Die Basis und den größten Anteil aller Evaluationskriterien stellt das Material dar. Ein Schachmotor ist immer bemüht die Materialbilanz für sich zu verbessern, ihm „Raum- und Zeitkriterien[12]“ beizubringen ist nicht sehr einfach. Um die Materialbilanz berechnen zu können, werden alle auf dem Brett befindlichen weißen Figuren mit einer entsprechenden Figurenbewertung (z.B. Bauer=100, Springer=300, Läufer=315, Turm=450, Dame=900)[13] aufsummiert und davon die schwarzen abgezogen.

```
material = ∑weisse Figur * Figurenwert - ∑schwarze Figur * Figurenwert
```

Mobilität

Ein weiteres markantes Kriterium ist die Mobilität. Figuren stehen auf dem Brett besser, wenn sie viel Bewegungsmöglichkeiten besitzen. Zu beachten ist aber die Tatsache, dass eine frühe Damenentwicklung, die Mobilität zwar sehr schnell erhöht, aber schachtheoretisch nicht zu vertreten ist. Daher sollte im Eröffnungsstadium eine Spiels davon abgesehen werden die Dame in dieses Kriterium mit aufzunehmen.

Die Berechnung erfolgt widerum sehr einfach:

```
mobilität = ∑ weisse Zugmöglichkeit - ∑ schwarze Zugmöglichkeit
```

Nun muss noch mit einem weiteren Gewichtungsfaktor festgelegt werden, in welchem Verhältnis das Material zur Mobilität steht. Z.B. könnte das Material doppelt so hoch gewichtet sein. Das Problem gute

[11] Um Experimente mit TD durchzuführen, musste ich in 6 Wochen die Basis für einen komplett neuen Schachmotor schreiben, da der bis dato entstandene *FUSc#* eine sehr langsame arraybasierte Brettrepräsentation besaß. Demzufolge brachten die ersten Tests mit diesem Motor sehr schlechte Ergebnisse die darauf zurückzuführen waren, dass er auch mit sehr gut eingestellten Parametern fast jede Partie verlor (das rating lag bei ca. 1600). Der komplett neue Schachmotor ist wesentlich schneller und basiert auf den RotatedBitBoards. Das gesamte *FUSc#*-Team war anschließend mit der neuen Engine beschäftigt.

[12] Schon die ersten großen Schachmeister erkannten, dass das Schachspiel nicht nur darin besteht, Material zu erobern (ausgenommen das Mattsetzen), sondern dass noch weitere wichtige Aspekte zu betrachten sind. Nach der Schachtheorie unterteilt sich die Bewertung in Material, Raum und Zeit, wobei der Raum die guten Manövrierungsmöglichkeiten für eine Partei und die Zeit eine schnelle Angriffswelle mit Initiative als Kompensation für Material bedeuten.

[13] Diese Werte werden im Schachmotor *FUSc#* verwendet. Kasparov gibt die folgenden an: Bauer=1, Springer=3, Läufer=3 ,Turm=4.5 und Dame=9 [Stein].

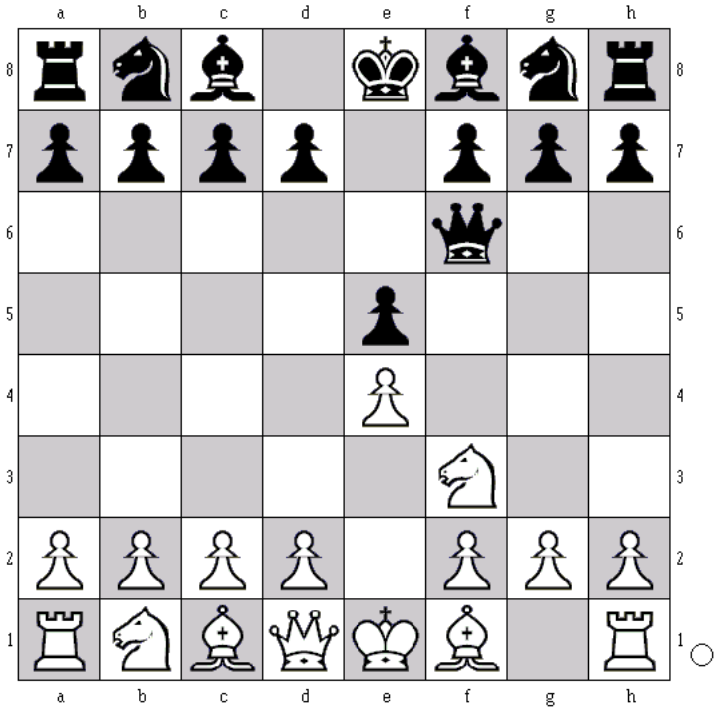

Abbildung 1.4: Typische Computerstellung, bei dem die Dame in das Mobilitätskriterium aufgenommen wurde

Gewichtungsfaktoren zu finden ist sehr groß. Nachdem ein Schachmotor programmiert wurde geht erst die lange Arbeit des Einstellens der Bewertungsfunktionen los. Bei professionellen Schachprogrammen werden oft Großmeister engagiert, die bei Testspielen Anregungen geben, inwieweit das Programm bestimmte Stellungsfaktoren über- oder andere unterschätzt. Beispielsweise wurden beim Match zwischen Kasparov und *Deep Blue* einige Schach-Großmeister zur Bewertungseinstellung engagiert. Je komplexer die Evaluationmethode ist, desto schwieriger kann diese Arbeit sein.

Ziel dieser Arbeit ist es gerade, eine Vorgehensweise anzugeben, die es dem Schachmotor ermöglicht das Problem selber zu lösen und die Bewertungsparameter eigenständig anzupassen.

Figurenqualitäten

Der größte Bereich der Stellungsbewertung stellt die Figurenqualität dar. Dabei können beliebige Stellungsmuster betrachtet werden. Beispielweise können bei den Bauern die Stellungseigenschaften isolierte Bauern, Doppelbauern und Freibauern, evaluiert werden. Diese Stellungsmuster repräsentieren positive und negative Eigenschaften einer Stellung. Beispielsweise hat es sich im Laufe der Jahre gezeigt, dass ein Doppelbauer in den meisten Partien eine Schwäche darstellte und demnach zu einem Abzug in der Bewertung führen sollte.

Problematik

Schachmotoren bewerten eine Reihe von Stellungseigenschaften. Einige stellen positive andere negative Eigenschaften einer Stellung dar. Nun ist es aber schachlich nicht gerechtfertigt, jedes Stellungsmuster gleich zu bewerten. Es gibt Stellungseigenschaften wie z.B. das Materialverhältnis und die Königssicherheit, die kaum durch andere kompensiert werden können. Deshalb liegt den einzelnen Faktoren ein Gewichtungsparameter bei. Da sich die Schachmotoren bei der Implementierung der einzelnen Stellungsmuster stark unterscheiden, gibt es keine Regel für die Wahl der Gewichte. Bei professionellen Schachmotoren

werden Großmeister zum justieren der Gewichte engagiert. Durch Partie- und Stellungsanalysen kann der Großmeister darauf hinweisen, welche Muster zu stark oder zu schwach in die Bewertung eingehen und geändert werden müssen. Die beste Lösung wäre, wenn ein Schachmotor eigenständig erkennt, welche Koeffizienten angepasst werden müssen.

1.2.2 Eröffnungsdatenbank

Nach vielen Jahren der Schachgeschichte haben sich Eröffnungssysteme als Klassifikation der ersten Züge von der Startstellung etabliert. So sind Zugfolgen, die auf den ersten Blick für eine Partei riskant erscheinen doch besser für diese, was sich oft durch langfristige Pläne zeigt.

statisch

Ein Schachmotor mit einem begrenzten Horizont hat nicht die Möglichkeit das alles zu erfassen, darum bedient man sich einfach einer Datenbank, in der die Schacherkenntnisse der letzten Jahrhunderte gespeichert sind und gibt dem Schachmotor die Möglichkeit diese zu verwenden. Wobei die Zugwahl meist deterministisch erfolgt. Eine bestimmte Zugfolge, die in der Datenbank zu finden ist, liefert immer einen bestimmten Folgezug. Um eine variable Spielweise zu erhalten, kann die Datenbank zu einer konkreten Stellung mehrere Folgezüge haben und einen mit einer bestimmten Wahrscheinlichkeit auswählen.

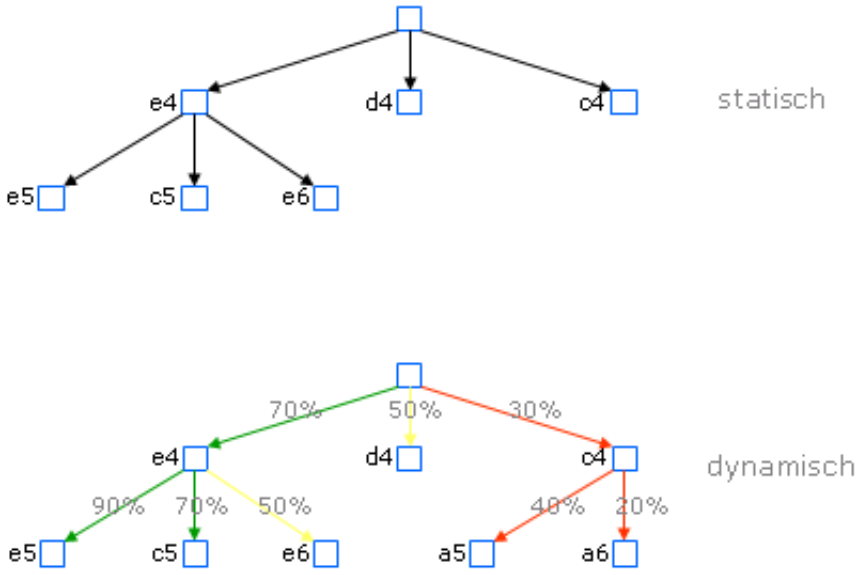

Abbildung 1.5: Eröffnungsbuch statisch / dynamisch

Im statischen Fall ist ein Buch mit festen Zugfolgen vorgegeben, der Schachmotor spielt demnach deterministisch oder kann mit einer gewissen Wahrscheinlichkeit einen Zug bei Alternativen wählen. Im dynamischen Fall erweitert sich das Buch ständig mit Wissen über eigene Erfolge und Misserfolge in Partien. Diese könnten z.B. prozentuall angegeben werden. Schachstellungen die einem Großmeister „liegen“ müssen nicht mit der gleichen Qualität von einem Schachprogramm angegangen werden... Da eine Veränderung der Spielweise durch das Anpassen der Zugwerte in der Zeit stattfindet, kann diese Eröffnungsbuchform auch *evolutionär* genannt werden.

dynamisch (evolutionär)

Es gibt Stellungen die ein Schachmotor „nicht versteht“. D.h. die Stellungskriterien erlauben es dem Schachmotor nicht, den schachmeisterlichen Hintergrund gewisser Stellungen korrekt zu analysieren und

zu bewerten. Daher ist es sinnvoll die vom Schachprogramm ausgewählten guten Wege höher zu bewerten und nach Niederlagen Wege mit negativen Gewichten zu belegen, so dass das Schachprogramm mit der Zeit an Erfahrung gewinnt und die für ihn besseren Stellungen zu forcieren versucht.

1.3 Mensch gegen Maschine

Ein Schachmeister findet sich auf dem Schachbrett in einer konkreten Schachstellung zu Recht und erkennt logische Zusammenhänge und kann diese in der Zeit (nachfolgenden Zügen) logisch fortsetzen, also z.B. den besten Zug finden. Er konkretisiert anhand der Muster einen logischen Plan, der Schwächen und Stärken der beiden Spieler berücksichtigt. Solch ein Plan kann sehr langfristig sein und nicht als reine Zugfolge, sondern eher als abstrakte Weiterführung der Partie, betrachtet werden.

Ein Schachprogramm spielt anders. Nach einer bestimmten Bewertungsfunktion versucht es eine Stellung zu forcieren, die für ihn am aussichtsreichsten scheint. Das heißt, es betrachtet eine bestimmte Zugfolgenlänge und bewertet die resultierenden Stellungen. Ein Schachprogramm zielt also einfach nur auf eine bessere Figurenstellung ab und spielt nicht nach einem Plan. Gewinnstrategien gegen Schachprogramme lauten demnach, den Spielplan langfristig anzulegen, taktische Stellungen möglichst zu vermeiden, da gerade dort die Stärken des Schachprogramms liegen.

Zusammenfassung der 10 goldenen Regeln beim Spielen gegen ein Schachprogramm [Stein]:

1. Ruhige Eröffnung wählen, möglichst früh von den Hauptvarianten abweichen

2. Taktische Verwicklungen vermeiden

3. Angriffsvorbereitungen sollten ruhig getroffen werden.

4. Nicht spekulativ spielen.

5. Positionelle Opfer für langfristige Ziele können sehr effektiv sein.

6. Horizonteffekte[14] sollten provoziert werden.

7. Kurzfristiger Materialgewinn wird kaum möglich sein

8. Ein unachtsamer Zug kann aus dem planlosen Spiel des Computers ein taktisches machen

9. Bei unklarem Mittelspiel lieber schnell ins Endspiel abwickeln. Dort spielen Computer am schwächsten.

10. Niemals in Zeitnot geraten.

Eine sehr eindrucksvolle Partie, die Garry Kasparov 2003 gegen *Fritz* spielte zeigt, dass der Weltmeister gegen eine Maschine mit typischen Anticomputerschachmitteln aggierend, problemlos gegen diese gewinnen kann.

1. Sf3 - Sf6 **2.** c4 - e6 **3.** Sc3 - d5 **4.** d4 - c6 **5.** e3 - a6 **6.** c5 - Sbd7 **7.** b4 - a5 **8.** b5 - e5 **9.**
Da4 - Dc7 **10.** La3 - e4 **11.** Sd2 - Le7 **12.** b6 - Dd8 **13.** h3 - O-O **14.** Sb3 - Ld6 **15.** Tb1 - Le7
16. Sxa5 - Sb8 **17.** Lb4 - Dd7 **18.** Tb2 - De6 **19.** Dd1 - Sfd7 **20.** a3 - Dh6 **21.** Sb3 - Lh4 **22.**
Dd2 - Sf6 **23.** Kd1 - Le6 **24.** Kc1 - Td8 **25.** Tc2 - Sbd7 **26.** Kb2 - Sf8 **27.** a4 - Sg6 **28.** a5 - Se7
29. a6 - bxa6 **30.** Sa5 - Tdb8 **31.** g3 - Lg5 **32.** Lg2 - Dg6 **33.** Ka1 - Kh8 **34.** Sa2 - Ld7
35. Lc3 - Se8 **36.** Sb4 - Kg8 **37.** Tb1 - Lc8 **38.** Ta2 - Lh6 **39.** Lf1 - De6 **40.** Dd1 - Sf6 **41.** Da4 - Lb7 **42.**
Sxb7 - Txb7 **43.** Sxa6 - Dd7 **44.** Dc2 - Kh8 **45.** Tb3 - ***AUFGABE***

[14] Ein Schachmotor, der einen Suchbaum der Tiefe *t* verwendet und durch einen „Nicht-Schlag-Zug“ anschließend forciert z.B. eine Figur verliert, hat manchmal das Problem, dass durch einen Zwischenzug diese Erkenntnis verloren gehen kann und das Problem über den Horizont *t* geschoben wird, da nach der Suchtiefe *t* nur noch die Schlagzüge in Betracht gezogen werden (siehe Ruhesuche).

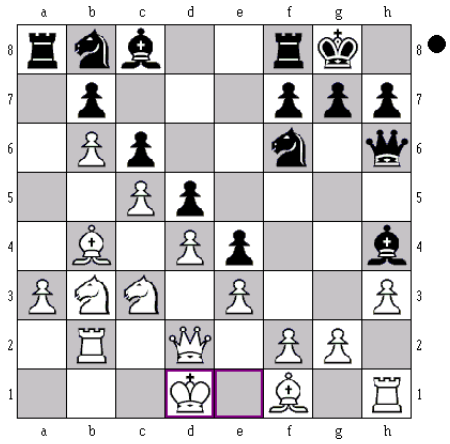

Abbildung 1.6: Garry Kasparov - X3D Fritz (2003), Spiel 3 nach **23.** Kd1-....

Kasparov zeigt eindrucksvoll gegen *Fritz*, eines der besten aktuellen Schachprogramme, dass er mit typischen langfristigen Manövern und einem geschloßenem Zentrum, dieses Programm deklassieren kann. Die planlose Führung der schwarzen Figuren und die sehr langfristigen Manöver von Weiß (beispielsweise begibt sich der König in aller Ruhe vom Zentrum zum Damenflügel) zeigen den großen Unterschied zwischen Mensch und Maschine.

Kapitel 2

Reinforcement Learning

Die Künstliche Intelligenz unterscheidet *überwachtes* und *unüberwachtes Lernen*. Unter überwachtem Lernen versteht man ein System (auch Agent genannt), dass in der Trainingsphase (von einem Lehrer) ein Tupel aus Input und gewünschtem Output erhält und versucht, die Überführung von den Input zu den Output-Werten, was einer Funktion entspricht, bestmöglich zu approximieren. In der anschließenden Testphase bekommt das System nur noch die Input-Werte und sollte den dazugehörigen Output eigenständig liefern. Anders ist es beim unüberwachten Lernen. Hier bekommt das System lediglich eine Menge von Input-Werten und soll eigenständig eine gute Klassifizierung der Daten finden. Beispielsweise versuchen Expectation Maximisation-Algorithmen dieses, indem sie ähnlichen Werten, die in einer bestimmten Nähe zueinander stehen, eine eigene Klasse zuordnen.

Beim *Reinforcement Learning* gibt es keinen Lehrer und das Ziel muss auch nicht bekannt sein. Die einzige Möglichkeit für den Agenten besteht nun aus Aktionen in seiner Umwelt. Diese Aktionen führen zu neuen Situationen, die wiederum Aktionen erfordern. Die Umwelt gibt dem Agenten ab und zu ein Signal (Reinforcementsignal) wie, „das war gut“ oder „das war schlecht“. Nun muss der Agent selbstständig Rückschlüsse auf die Aktionen der Vergangenheit ziehen und damit entscheiden, wie er in Zukunft auf eine ähnliche Situation wieder reagieren würde. Eine negative Rückmeldung für eine konkrete Situation kann auch eine Bewertung der Situationen, die zu dieser führten, verändern. Dies gilt auch in der Umkehrrichtung. Ein Agent kann viele Situationen durchlaufen, ohne eine Rückmeldung zu erhalten. Beispielsweise erhält ein Agent, der ein Spiel lernen möchte, erst am Ende eine Meldung (Sieg, Unentschieden oder Niederlage).

Die beste Strategie eines Agenten ist nun keineswegs, immer die beste Aktion in einer Situation zu wählen, die ihm gerade zur Verfügung steht. Er sollte auch weniger gute, oder nicht bekannte ausprobieren und damit mehr über seine Umwelt in Erfahrung bringen. „*Daraus ergibt sich, dass Agenten die immer oder niemals weiterforschen stets scheitern.*“ [Luger]. Dieses Verhalten wird der Agent im Laufe des Lernvorgangs verringern und gegen eine gute Strategie konvergieren.

Da das Reinforcement Learning in keine der beiden genannten Klassen einzuordnen ist, stellt es eine eigene Lernklasse dar. [Zhang] benennt diese Klasse als *halbüberwachtes Lernen*. Es sei erwähnt, dass es darüber unterschiedliche Auffassungen gibt, beispielsweise ist bei [Luger] nachzulesen, dass er Reinforcement Learning zu dem unüberwachten Lernen zählt.

2.1 Reinforcement Learning-Strategien

Es gibt 3 strategische Ansätze, auf denen sich praktisch alle Algorithmen begründen, die zu Reinforcement Learning gezählt werden können [Luger].

Dynamische Programmierung

Dynamische Programmierung berechnet Wertefunktionen, indem sie Werte von Nachfolgerzuständen auf Vorgängerzustände übertragen. Beim Einsatz werden die einzelnen Zustände anhand eines Modells der nächsten Zustandsverteilung systematisch nacheinander aktualisiert. Die Realisierung von RL durch Dynamische Programmierung basiert auf folgender Gleichung:

$$V^{\pi}(s) = \sum_a \pi(a|s) * \sum_{s'} \pi(s \to s'|a) * (R^a(s \to s') + \gamma(V^{\pi}(s')))$$

Monte Carlo-Methoden

Bei den Monte Carlo-Methoden gibt es ähnlich wie bei der *Temporalen Differenz* eine update-Funktion. Diese wird aber erst nach Erreichen eines Endzustandes (beim Schach, wäre es das Partieende), angewandt. Bei Monte-Carlo-Methoden ist es nicht nötig, die gesamte *Umwelt* zu kennen, sondern lediglich Auszüge dieser (Sequenzen), die dann repräsentierend für die Umwelt, Wahrscheinlichkeiten der Zusammenhänge liefern.

Temporale Differenz-Methoden

TD stellen eine Kombination der MC-Idee und dynamischer Programmierung dar. Wie das MC-Verfahren kann es direkt mit grober Erfahrung ohne ein Modell der Umweltdynamik lernen, es erfordert keine komplette Umwelt. Der Vorteil dabei ist aber, dass nicht wie bei MC der Zyklus einmal durchlaufen werden muss, um die Parameter aktualisieren zu können. Es kann je nach Problemstellung schon im nächsten Zustand eine Aktualisierung des vorhergehenden vorgenommen werden. Trotzdem konvergiert TD zu einer optimalen Strategie. Wie bei der dynamischen Programmierung aktualisiert TD die Zustandswerte (oder Evaluierungsfunktion) anhand der bereits gelernten Zustände, ohne zu wissen, wie das Ziel aussieht. Eine spezielle Variante ist das $Q-Lernen$, wobei $Q : (Zustand \times Aktion) \to Wert$, die Überführungsfunktion von Zustand-Aktions-Paaren zu gelernten Werten ist. Für das Q-Lernen gilt folgender Aktualisierungsschritt, mit $c, \gamma = 1$ und $r_{t+1} = Belohnung bei s_{t+1}$:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + c * [r_{t+1} + \gamma * max_a(Q(s_{t+1}, a_{t+1}) - Q(s_t, a_1))]$$

Für den Einsatz von Reinforcement Learning-Strategien beim Schach bietet sich die Temporalen Differenz-Methoden an, da zum einen die Komplexität des Schachspieles den Raum aller möglichen Brett-Konfigurationen sehr groß macht und zum anderen eine Evaluierungsfunktion (Bewertungsfunktion) existiert, die es erlaubt verschiedene zeitliche Zustände in Relation zu setzen. Mit der Vorgabe Schachpartien zu gewinnen und sowohl den Gewinn- als auch Verlustweg als solchen zu erkennen, kann ein Agent beim spielen und anschließendem aktualisieren der Bewertungs-Koeffizienten sein Spiel verbessern.

Beispiel: Tic-Tac-Toe

Im Zusammenhang mit Temporaler Differenz wird sehr oft das Beispiel Tic-Tac-Toe erwähnt. Es soll auch hier nicht fehlen. Zunächst wird eine Wertetabelle (value-function-table), deren Einträge für jede mögliche Kombination einer Stellung und deren Positionsbewertung (state current value) angelegt. Die Werte liegen im Intervall $[0,1]$, wobei eine 1 den Sieg von Spieler 1 und eine 0 den Sieg von Spieler 2 bedeutet. Die Initialwerte liegen zum Startzeitpunkt bei 0.5 für alle Positionen.

Der Agent (Spieler 1) beginnt zu spielen und benutzt die Wertetabelle um einen Zug zu wählen. Dabei wird er meistens den besten für sich wählen. Ab und zu sollte er aber auch forschende Züge unternehmen und damit das Wissen der Funktion zu verbessern. Nachdem der Agent nun eine Partie gespielt und diese z.B. gewonnen hat, wird er die Bewertung der letzten Stellung der Partie in der Wertetabelle vergrössern. Das macht er auch mit der Stellung davor, aber um einen kleineren Faktor, usw. (siehe Abbildung 2.1). Sei s_t der Zustand vor einem gewählten Zug und s_{t+1} der Zustand danach, dann ist der Wert $V(s_t)$ wie folgt aktualisiert:

$$V(s_t) \leftarrow V(s_t) + \alpha * [V(s_{t+1}) - V(s_t)]$$

α ist ein kleiner positiver Wert (step-size-parameter) kleiner 1, der die Lernrate bestimmt und nach jedem Aktualisierungsschritt kleiner wird und schliesslich gegen 0 konvergiert. Die temporale Differenz zwischen den Bewertungen der Stellungen s_t und s_{t+1} wird mit $V(s_{t+1}) - V(s_t)$ berechnet, dadurch erhält das Verfahren auch seinen Namen.

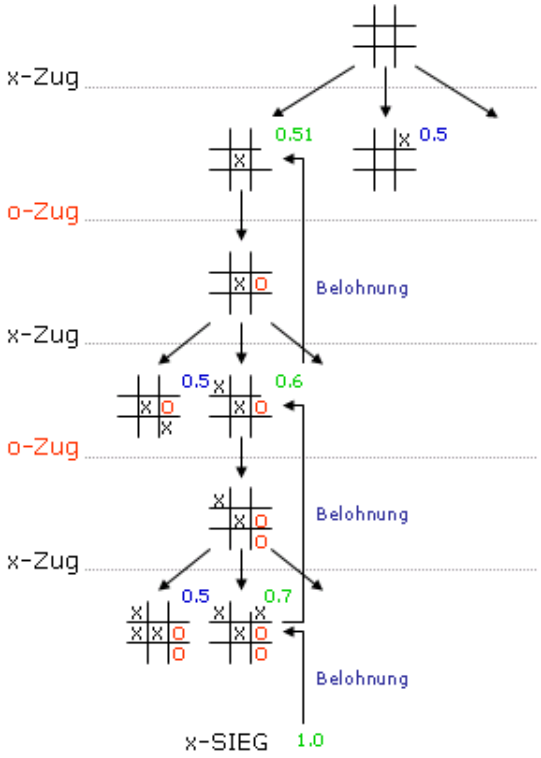

Abbildung 2.1: Verwendung von TD bei Tic-Tac-Toe

Dieses Beispiel [nach Luger] zeigt den Einsatz von Temporaler Differenz bei TicTacToe. Vor dem ersten Spiel wurden alle möglichen Stellungen als Remisstellungen gewertet, der Agent besitzt demnach keine gute Spielstrategie. Beginnend bei der Startstellung wurde nun eine mögliche Partie nachgestellt, die von oben nach unten zu lesen ist. Der letzte X-Zug (in der Mitte) baut eine Doppeldrohung auf, die O nicht parrieren kann und nach seinem nächsten Zug, egal welchen, die Partie verliert. Der Agent bekam als Rückmeldung nun eine positive Bewertung (SIEG) und wird der Zielstellung eine bessere Bewertung zukommen lassen. Die Stellung mit X am Zug vor dieser, wird ebenfalls etwas positiv verstärkt. Möglicherweise werden alle Bewertungen der eigenen Stellungen auf diesem Weg positiv verstärkt, aber je größer der Abstand zur Endstellung ist, desto geringer sollte die Veränderung sein. Im Beispiel wurde die Siegstellung mit 1 bewertet und demnach könnte die vorherige Stellung mit 0.7 bewertet werden.

2.2 Temporale Differenz in der Schachprogrammierung

Im folgenden Abschnitt wird versucht, den Einsatz von Temporaler Differenz (TD) in Schachprogrammen historisch nachzuvollziehen. Es ist sehr wahrscheinlich, dass der Einsatz von TD in einigen weiteren Schachprogrammen Verwendung gefunden hat, aber diese nicht ausdrücklich zur Weitererforschung auf diesem Gebiet beigetragen haben oder die Ergebnisse nicht veröffentlicht wurden und deshalb nicht aufgeführt werden. TD wurde auch beim Chinesischen Schach eingesetzt [Thong].

2.2.1 Der erste Versuch - SAL (Search and Learning)

Entwickelt und getestet wurde *SAL* von Michael Gherrity [Gherrity]. Die Struktur von *SAL* erlaubt es, einen Zuggenerator für verschiedene Spiele zu schreiben (das beinhaltet den Regelsatz des jeweiligen Spieles) und anschliessend mit einem Suchbaum den besten Zug zu spielen. So wurden Zuggeneratoren für Tic-Tac-Toe, 4 gewinnt und Schach implementiert. Im Laufe mehrerer Spiele erlernt *SAL* "gute und schlechte" Züge. Die Evaluation der einzelnen Züge übernimmt ein Neuronales Netz. TD wurde für die Parameteroptimierung des Netzes verwendet, wobei die Evaluationswerte der Wurzelknoten des Suchbaumes verglichen wurden. Im einem Vergleich mit dem Schachprogramm *GNUChess*[1] in 4200 absolvierten Partien, wobei *SAL* 1031 Stellungsbewertungsfaktoren benutzte, erreichte es 8 Remis und verlor den Rest. Im Durchschnitt schaffte *SAL* eine Suchtiefe von 4 und benötigte 1500 Knoten.

Gründe für den Misserfolg waren zum einen der zu starke Gegner und die daraus ergebene Konsequenz, dass jede Stellung scheinbar zur Niederlage führt und zum anderen die unzureichende Ausstattung des Programmes, was aktuelle Schachprogrammierstandards betrifft. Das Ziel von Michael Gherrity war es mit *SAL* ein Programm zu entwerfen, das verschiedenste Spiele lernen kann. Daher wurden schachspezifsche Lösungen, wie z.B. Transpositionstabellen und Zugwahlheuristiken, nicht implementiert. Der Ansatz stimmt jedoch optimistisch.

2.2.2 NeuroChess

Das von Sebastian Thrun entwickelte Schachprogramm *NeuroChess* verwendet ein Neuronales Netz als Evaluationsfunktion und eine TD-Methode, basierend auf den Wurzelknoten, um die Koeffizienten zu ändern [Thrun]. Im Gegensatz zu *SAL*, spielte *NeuroChess* in der Lernphase nur gegen sich selber. Wobei es nicht nur Spiele von der Startstellung aus spielte (10%), sondern auch Stellungen aus Großmeisterpartien (90%). Spätere Experimente mit einem anderen Programm zeigten, dass das Spielen gegen sich selbst keine guten Ergebnisse bringt. „*... we have found self-play to be a poor way of learning.*" [BaxTriWea2].

In den Experimenten mit *GNUChess*, wobei beide Programme bis zur Tiefe 2 ohne Ruhesuche und identischer Evaluation gerechnet haben, konnte *NeuroChess* mit den gelernten Koeffizienten, nur $13,1\%$ der Partien gewinnen (316 von 2400).

Thrun gab zwei fundamentale Probleme bei seinem Ansatz an, zum einen die eingeschränkte Trainingszeit: „*... and it is to be expected that excellent chess skills develop only with excessive training time.*"[Thrun] und zum anderen, dass bei jedem Lernschritt (Bewertungsupdate nach einer Partie) Informationen verloren gehen: „*... because the features used for describing chess boards are incomplete...*" [Thrun]. Als Fazit gibt er an: "*It is therefore unclear that a TD-like approach will ever, for example, develop good chess openings.*" [Thrun]

2.2.3 Deep Blue

Gerald Tesauro beschreibt eine bessere Möglichkeit der Verwendung von TD zum Einstellen der Evaluationskoeffizienten [Tesauro2]. Dazu verwendet er *SCP*[2] und zeigt experimentell, dass bei einer Suchtiefe von 1 mit TD keine guten Ergebnisse zu erreichen sind. Anders ist es, wenn TD die Suchtiefe 1 in Verbindung mit der Ruhesuche verwendet. Aus vorher „guten" per Hand eingetragenen Koeffizienten, ergaben sich Koeffizienten, denen er eine „*high-quality*" [Tesauro2] zusprach.

Eine Version dieses Algorithmus wurde auch bei *Deep Blue*[3] verwendet, zum größten Teil zum Verbessern der Königssicherheit. Die Trainingsphase benutzte dabei die Tiefe 4 und anschließender Ruhesuche.

[1] Ein *open source* Programm, geschrieben in C. Die Spielstärke zum Zeitpunkt des Vergleichkampfes ist der menschlichen Meisterstärke entsprechend.

[2] Ein einfaches *open source* Schachprogramm (1987) mit dem Vorteil, dass Suche und Evaluation im Programm relativ gut getrennt sind. Das war der Hauptgrund für Tesauro gewesen, denn er gab an, dass es sehr wichtig sei, eine stabile Suchprozedur zu besitzen, während es die Koeffizientengewichte trainiert. SCP arbeitet mit einem AlphaBeta-Algorithmus fester Suchtiefe und evaluiert 165 Faktoren.

[3] Von IBM entwickelter Schachmotor, der den Weltmeister im Schach geschlagen hatte.

1997 spielte der damalige Weltmeister Garry Kasparov gegen den von IBM (in 5 Jahren) entwickelten Schachmotor *Deep Blue* und verlor das Match mit 2.5 zu 3.5 Punkten. Das erste Match der beiden 1996 konnte Kasparov mit 4 zu 2 für sich entscheiden, wobei er nur eine Partie verlor. Dieser Schachmotor basierte auf dem Forschungsprojekt *Deep Thought* der Carnegie Mellon Universität. Speziell für *Deep Blue* wurde ein Chip entwickelt, der 2 Millionen Stellungen pro Sekunde verarbeiten konnte. Der Gesamtrechner umfasste 220 dieser speziellen Chips, die parallel arbeiteten. Nach dem Match wollte Kasparov ein Revanchekampf gegen *Deep Blue* haben, doch IBM zerlegte den Rechner.

Als Erfolg konnte die Tatsache angesehen werden, dass im Rematch 1997 gegen Garry Kasparov an einigen kritischen Stellen die „optimierten“ Koeffizienten *Deep Blue* veranlassten sehr starke, positionelle Züge zu spielen, von den Kasparov später behauptete, sie seinen von einem Menschen gespielt und möglichenweise Betrug unterstellte. Eine dieser Stellungen ist in Abbildung 2.2 zu sehen.

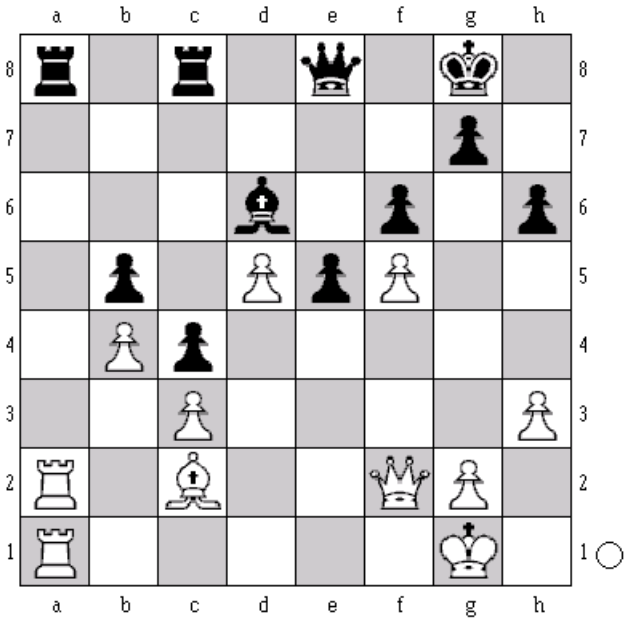

Abbildung 2.2: Deep Blue - Garry Kasparov, Spiel 2 beim Rematch 1997.

In dieser Stellung hätte Deep Blue mit den normalen Koeffizienten Db6 gespielt, was ein typischer Computerzug gewesen wäre, denn er hätte auf einen Bauerngewinn spielen können. Kasparov und das Publikum waren nun sehr überrascht, als Deep Blue Le4! spielte, ein sehr positioneller Zug, und damit jegliches Gegenspiel erstickte. Die Drohung Db6 ist nun sehr viel stärker. Kasparov verlor diese Partie schließlich.

2.2.4 Temporale Differenz und Backgammon

Der Erfolg von Tesauros TD-Gammon

Gerald Tesauro veröffentlichte 1995 einen Artikel über die Verwendung von TD in seinem Backgammonprogramm [Tesauro3]. Das Programm trainierte die Koeffizienten der sehr komplexen Bewertungsfunktion mit dem Algorithmus TD(λ). Die Stärke des Programmes war anschliessend sehr hoch. Das erste Debut 1992 bei den Backgammon-Weltmeisterschaften spielte eine Version, die zuvor 800.000 Partien trainiert hatte. Das Programm verlor von den insgesamt 38 Turnierpartien lediglich 7. Eine Version, die 1.500.000 Partien trainiert hatte, verlor gegen Bill Robertie in 40 Testpartien sogar nur eine.

Anzumerken ist, dass Bill Robertie (Abbildung 2.3) einer der stärksten Backgammonspieler der Welt ist. Er hat den Titel des Weltmeisters in Monte Carlo 1983 und 1987 gewonnen, sowie das Turnier auf den Bahamas 1993 und den Istanbul Weltcup 1994.

Abbildung 2.3: Backgammonweltmeister: *Bill Robertie* (aus [Robertie])

TD(λ)

TD wurde das erste mal von A.L. Samuel 1959 beschrieben [Samuel] und später von R.Sutton 1988 weiter formalisiert [Sutton]. Der TD(λ)-Algorithmus wird im folgenden anhand des Spielproblems Schach, basierend auf [BaxTriWea] diskutiert.

Sei S die Menge aller möglichen Schachstellungen (Zustände) und t der Zeitpunkt ($x_t \in S$ das zugehörige Brett), der nach einer Folge von t ausgeführten Aktionen auf dem Brett entsteht. Der Einfachheit halber wird angenommen, dass jede Partie eine feste Länge von N Zügen besitzt. Jede Schachstellung besitzt eine Liste von Zugmöglichkeiten (Aktionen) A_{x_t}, die in eine andere Schachstellung übergehen. Im Schach sind das die legalen Züge einer Seite. Der Agent wählt eine Aktion $a \in A_{x_t}$ aus und damit den Zustandsübergang x_t zu x_{t+1} mit einer Wahrscheinlichkeit $p(x_t, x_{t+1}, a)$ (Abbildung 2.4). Zu x_t stellt der Folgezustand x_{t+1} die Brettposition dar, die entsteht, wenn ein kompletter Zug ausgeführt wurde, also auch die Antwort des Gegners. Folglich werden nur die eigene Stellungen untersucht, also diejenigen, an denen der Agent eine Entscheidung (Aktion) treffen muss.

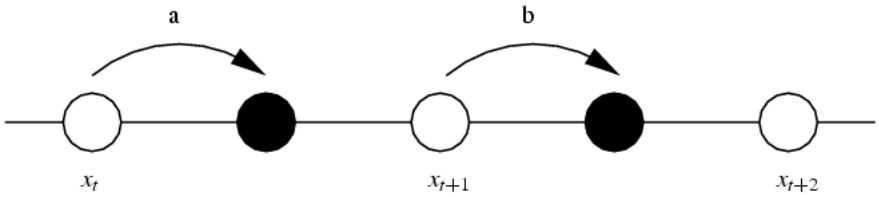

Abbildung 2.4: Zustandsübergänge

Nach einer gespielten Partie bekommt der Agent einen Rückgabewert (Reward) $r(x_N)$, der das Resultat wiederspiegelt (beispielsweise 0 für Remis, 1 für Sieg und -1 für eine Niederlage). Der Erwartungswert der Belohnung bei einer idealen Bewertungsfunktion $J^*(x)$ ist $J^* := E_{x_N|x} r(x_N)$. Das Ziel der Lernverfahrens ist es, die ideale (wohl nicht lineare Bewertungsfunktion) $J^*(\cdot) : S \to \mathbb{R}$ durch eine lineare Funktion zu approximieren. Dazu wird eine parametrisierte Klasse von linearen Funktionen $J' : S \times \mathbb{R}^k \to \mathbb{R}$ betrachtet. Gesucht ist der Parametersatz[4] $\omega = \omega_1, ..., \omega_k$, der die Funktion $J^*(\cdot)$ bestmöglich approximiert.

Um die Koeffizienten verändern zu können, muss zunächst eine Partie $x_1, ..., x_{N-1}, x_N$ mit festem ω gespielt werden. Der TD(λ)-Algorithmus berechnet die temporalen Differenzen der Bewertungen, der in einer Partie erreichten Stellungen

[4] Also der Koeffizientenvektor für die Evaluationsfaktoren.

$$d_t := J'(x_{t+1}, \omega) - J'(x_t, \omega)$$

Da die letzte Stellung x_N mit $J'(x_N, \omega) = r(x_N)$ gleichzusetzen ist, ergibt sich für $d_{N-1} = r(x_N) - J'(x_{N-1}, \omega)$. Es wird von einer idealen Bewertungsfunktion erwartet, dass zu einem Zeitpunkt t, indem die Stellung positiv für den Agenten ist, auch zu einem Sieg der Partie führt. Sollte die temporale Differenz zwischen x_{t+1} und x_t positiv sein, so hat sich die Stellung des Agenten verbessert. Das kann, da ein Zug des Gegners dazwischen lag, darin liegen, dass dieser einen Fehler gemacht hat. Daher sind positive Differenzen mit Vorsicht zu betrachten, denn Fehler des Gegners sollten nicht gelernt werden. Ergibt die Differenz einen negativen Wert, so wurde die Stellung x_t nicht richtig bewertet, da sie sich nach einem Zug als schlechter herausgestellt hat. Im folgenden muss die Bewertung der Stellung x_t negativer ausfallen, das heißt, es wird die kleinstmögliche Veränderung des Parametersatzes gesucht, der die größtmögliche Wirkung in diese Richtung besitzt. Um die Richtung und die Parameter zu finden, wird der Gradient $\nabla J'(\mathbf{.}, \omega)$ berechnet. Der Vektor ω wird wie folgt aktualisiert

$$\omega := \omega + \alpha \sum_{t=1}^{N-1} \nabla J'(x_t, \omega) \left[\sum_{j=t}^{N-1} \lambda^{j-t} d_j \right].$$

Der Gradient gibt die Richtung der Veränderung vor und Δt die Stärke.

$$\left[\sum_{j=t}^{N-1} \lambda^{j-t} d_j \right] =: \Delta t$$

Δt ist die gewichtete Summe der Differenzen im Rest der Partie. Für $\Delta t > 0$ gilt, dass die Stellung x_t vermutlich unterbewertet wurde, der Vektor ω wird mit einem positiven Vielfachen des Gradienten addiert und demnach ist die Bewertung der Stellung mit den aktualisierten Parametern höher als zuvor. Mit $\Delta t < 0$ wird die Stellung x_t als überbewertet interpretiert und folglich der Vektor mit einem negativen Vielfachen des Gradienten addiert. Der positive Parameter α kontrolliert die Lernrate und konvergiert schrittweise nach jeder Partie langsam gegen 0. Der Parameter λ kontrolliert dabei den Bezug der temporalen Differenzen einer Stellung x_t bis zum Ende der Partie. Wird $\lambda = 0$ gewählt, so werden die Folgestellungen nicht berücksichtigt, bei $\lambda = 1$ werden alle Stellungen bis zum Ende gleich betrachtet. [BaxTriWea] geben $\lambda = 0.7$ als heuristisch ermittelten Wert an.

Unterschied Backgammon und Schach

Da im Backgammon der Faktor Wahrscheinlichkeit eine große Rolle spielt, macht eine sehr tiefe Vorausberechnung der Möglichkeiten wenig Sinn. Um einen Zug zu bestimmen kann daher mehr Rechenzeit für eine komplexe Bewertungsfunktion verwendet werden, so z.B. mit einem sehr großen Neuronalen Netz. Man kann sagen, dass kleine Veränderungen der Stellungen im Backgammon zu kleinen Veränderungen der Bewertung führt. Im Schach bestimmt zum größten Teil die Taktik welcher Zug einer Stellung der beste ist. Es muss also mehr Rechenzeit darauf verwendet werden, die Zugmöglichkeiten der Spieler in Betracht zu ziehen und die erhaltenen Stellungen möglichst schnell zu bewerten. Es ist daher nicht sehr ratsam ein Neuronales Netz ausschließlich als Zugvorhersage zu benutzen.

Ein Neuronales Netz kann Probleme sehr gut klassifizieren [Rojas] und Stellungen zu Stellungstypen zusammenfassen, doch reicht die Veränderung einer einzigen Eigenschaft aus, um die Bewertung sehr stark zu verändern. Kleine Veränderungen können im Schach also eine sehr grosse Wirkung mit sich bringen. Eine Kombination aus Vorrausberechnung in einem Suchbaum und der in Tesauros Backgammonprogramm *TD-Gammon* verwendeten TD(λ)-Algorithmus scheint daher eine naheliegende Lösung zu sein.

2.2.5 KnightCap

KnightCap ist ein unter Linux und für den AP1000+ von Andrew Tridgell entwickelt und parallel arbeitendes Schachprogramm [Tridgell]. Es war vermutlich die Problematik der Stellungsbewertung und der Erfolg von Gerald Tesauros *TD-Gammon*, die Andrew Tridgell[5] zusammen mit Jonathan Baxter und Lex

[5] Zitat aus [Tridgell]: “Work on KnightCap is now concentrating on automatic learning algorithms to improve the search and evaluation functions ...”

Weaver motivierte, eine vorteilhaftere Nutzung von TD(λ) zu entwickeln [BaxTriWea] und damit die Evaluationsfunktion in einem Schachmotor zu lernen. Die Idee bestand darin, nicht auf den Wurzelknoten der Suchbäume zu arbeiten, sondern auf dem best forcierten Blattknoten einer Stellung. Dazu mussten einige Anpassungen vorgenommen werden, z.B. die Repräsentation der Evaluationsfaktoren in einem Vektor. Das erste Experiment brachte schon einen Erfolg. In nur 3 Tagen und 308 Spielen auf dem *Internet Chess Server(ICC)* verbesserte sich die Bewertung von *KnightCap* von 1650 auf 2150.

Struktur

Der Aufbau des Schachmotors *KnightCap* unterscheidet sich von dem in professionellen Programmen kaum. Alle aktuellen Standardtechniken wurden implementiert. Der grosse Unterschied zu anderen Programmen ist die Brettdarstellung. KnightCap arbeitet auf dem *ToPiecesBoard*. Damit ist eine Bewertungsfunktion möglich, die sehr schnell komplexe Muster identifizieren kann. MTD(f) [Plaat] wurde als Zugwahlalgorithmus verwendet. Die Bewertungsfunktion unterscheidet zwischen 4 Stellungstypen: Eröffnung, Mittelspiel, Endspiel und Mattstellungen. Jedem Stellungstyp stehen 1468 Bewertungsfaktoren zur Verfügung, was dazu führt, dass KnightCap 5872 Koeffizienten unterschiedlich bewerten kann. Die mit einem Eröffnungsbuch ausgestattete Version besitzt eine Spielstärke von ca. 2400 – 2500 Elopunkten und besiegt regelmässig Internationale Meister (Elo 2400 – 2600).

MinMax und TD(λ) werden zu TD-Leaf(λ)

Die Strategie, die Aktion *a* zu einer Stellung *x* zu wählen, die anschließend dem Gegner minimale Chancen zuspricht, wird bei *TD-Gammon* verwendet.

$$a(x) := argmin_{a \in A_x} J'(x'_a, \omega)$$

x_a ist die Stellung, die nach Ausführung der Aktion *a* in Stellung *x* entsteht.

Da im Schach diese Vorgehensweise nicht sehr vielversprechend ist (einen Zug voraus zu schauen) und durch einen Zugwahlalgorithmus alle Zugmöglichkeiten bis zu einer bestimmten Tiefe untersucht werden müssen, wurde der TD(λ)-Algorithmus in *KnightCap* so angepasst, dass nicht auf der Wurzel des Suchbaumes gearbeitet wird, sondern der best-forcierte Blattknoten als Bewertungsgrundlage genommen wird [BaxTriWea]. Der Algorithmus wird TD-Leaf(λ) genannt. Im folgenden sei $J'_d(x,\omega)$ der Evaluationswert zur Stellung *x*, der durch $J'(\cdot,\omega)$ berechnet in einer Suchtiefe *d* von *x* aus erreichbar ist. Dadurch ergeben sich folgende Änderungen für die in TD(λ) beschriebene Berechnung der temporalen Differenzen:

$$d_t := J'_d(x_{t+1},\omega) - J'_d(x_t,\omega)$$

und das Aktualisieren des Vektors ω

$$\omega := \omega + \alpha \sum_{t=1}^{N-1} \nabla J'_d(x_t,\omega) \left[\sum_{j=t}^{N-1} \lambda^{j-t} d_j \right].$$

TDLeaf(λ)-Algorithmus

Der TDLeaf(λ)-Algorithmus [BaxTriWea] arbeitet zusammenfassend nun wie folgt:

Sei $J(\cdot,\omega)$ eine Klasse von Evaluierungsfunktionen, parametrisiert mit $\omega \in \mathbb{R}^k$. Seien weiterhin $x_1,...,x_N$ N Stellungen, die während einer Partie betrachtet werden und $r(x_N)$ das Resultat. Als Konvention gelte: $J(x_N,\omega) := r(x_N)$.

1. Für jeden Zustand x_i, berechne $J_d(x_i,\omega)$ unter Anwendung der MinMax-Suche bis zur Tiefe d von x_i unter Verwendung von $J(\cdot,\omega)$, welches die Blätter evaluiert. Beachte, dass d von Stellung zu Stellung variieren kann.

2.Sei x_i^l das Blatt, welches ausgehend vom Knoten x_i durch die Hauptvariante erreicht wird. Sollte es mehrere Knoten geben, so wähle einen per Zufall. Beachte:

$$J_d(x_i, \omega) = J(x_i^l, \omega)$$

3.For $t = 1$ to $N-1$ berechne die temporalen Differenzen:

$$d_t := J(x_{t+1}^l, \omega) - J(x_t^l, \omega)$$

4.Update ω anhand der TD-Leaf(λ)-Formel:

$$\omega := \omega + \alpha \sum_{t=1}^{N-1} \nabla J(x_t^l, \omega) * \left[\sum_{j=t}^{N-1} \lambda^{j-t} d_j \right]$$

Versuche und Ergebnisse mit KnightCap

Eine kurze Zusammenfassung der Ergebnisse aus [BaxTriWea2].

Experiment 1

Alle Koeffizienten wurden mit 0 initialisiert, bis auf die Materialwerte (1 Bauer, 4 Läufer und Springer, 6 Turm, 12 Dame). Auf dem Internetschachserver FICS wurde die Stärke von 1650 von KnightCap zunächst in 25 Partien mit diesem Vektor ermittelt und anschliessend in 3 Tagen und 308 Spielen mit dem Einsatz von TD auf 2150 erhöht.

Experiment 2

Alle Koeffizienten wurden auf den Bauernwert gesetzt und anschließend 1000 Spiele auf ICC gespielt und mit TD gelernt. Die Bewertung stieg von 1250 auf 1550.

Experimente mit Selbsttraining

Es wurde versucht, die Koeffizienten zu lernen im Spielen gegen sich selbst (600 Partien). Diese Version konnte in 100 Partien nur 11% der Punkte gegen die in Experiment 1 erhaltene, holen. Dadurch kam man zu dem Schluss, dass das Spielen gegen sich selbst zu schlechteren Ergebnissen führt. Es sei angemerkt, dass Beal und Smith in einer anderen Arbeit zeigten, dass TD-Leaf(λ) und das Spielen mit sich selbst zu positiven Ergebnissen führen kann [Beal].

2.2.6 FUSc#

Der Informatik-Mathematik-Fachbereich der Freien Universität Berlin hat im Oktober 2002 im Rahmen eines Seminars die *AG Schachprogrammierung* gegründet aus der der Schachmotor *FUSc#* hervorging. Programmiert in C# und als Open Source Projekt realisiert, unterstützt *FUSc#* den Protokolltyp *Universal Chess Interface (UCI)*[6] und kann somit z.B. unter den Oberflächen von *Fritz* oder *Arena* verwendet werden. Die erste lauffähige Version spielte bei einigen Internetturnieren und auf dem Schachserver *http://www.schach.de*[7] und erreichte eine Bewertung von ca. 1700. Die Stellungsbewertung umfasste bereits einige hundert Stellungsfaktoren, was dem Programm auch eine relativ gute Spielqualität zusprach. Das Problem war die langsame arraybasierte Brettdarstellung und der noch etwas primitive Algorithmus. Zwar wurden einige Heuristiken in den Alpha-Beta-Algorithmus eingefügt, wie z.B. Killerzüge und die

[6] Ein Verweis dazu ist im Anhang zu finden.
[7] Auf diesem Server spielt *FUSc#* mit dem Pseudonym *deepfusch.*

Hauptvariante, doch gab es noch keine Optimierungen hinsichtlich Transpositionstabellen und Nullmovepruning.

Unter dem projektinternen Namen *DarkFUSc#* wurde im Januar 2003 eine neue Engine ins Leben gerufen, die zunächst als Clone in der Vorgängerversion untergebracht war, nun aber eine vollkommen, eigenständige Applikation darstellt. Im weiteren Verlauf bezeichnet *FUSc#* genau diesen neuen Motor. Als neue Brettdarstellung wurden die RotatedBitBoards gewählt, deren Verwendung eine enorme Performancesteigerung mit sich brachte. Stellungen in denen wir vorher in Tiefen von 4-5 rechneten können nun in 7-8 untersucht werden. Ebenfalls implementiert wurde neben den Transpositionstabellen auch die Nullmovetechnik. Die aktuelle Bewertung liegt bei ca. 1900.

TD-Leaf(λ) in FUSc#

Die ersten Versuche mit Temporaler Differenz in dem ersten *FUSc#*–Schachmotor brachten keine Erfolge mit sich. Das Programm hatte gegen anderen Schachprogramme nur sehr wenige Siege aufzuweisen, die für das Lernen mit TD unablässig sind. Das Programm muss sowohl negative, als auch positive Erfahrungen machen und die entsprechenden Faktoren besser oder schlechter bewerten. Da der Schachmotor fast alle Partien taktisch verlor, wurden auch Stellungen, in denen *FUSc#* positionell sehr gut stand, als schlecht bewertet.

Kapitel 3

Stellungsklassifikation

3.1 Schachspiel: Eröffnung, Mittelspiel, Endspiel

Als Schachanfänger lernt mal zunächst auch genau diese 3 Stellungstypen zu unterscheiden, damit man nicht die Dame oder den König zu früh ins Spiel bringt. In der *Eröffnungsphase* heißt es: Leichtfiguren entwickeln, König rochieren, Zentrum mit Bauern besetzen. Die *Mittelspielphase*, die am schwierigsten zu erlernen ist, erfordert beispielsweise das Öffnen von Linien, Besetzung dieser Linien mit Türmen, Angriff auf den gegnerischen König. Es gibt viele typische Konstellationen und Ziele. Im *Endspiel* ist es der König, der zur wichtigsten Figur wird. Er drängt den gegnerischen König beispielsweise von seinen eigenen Bauern ab, die sich im Laufe des Spieles dann (unter anderem) zu einer Dame umwandeln und eine schnelle Entscheidung bringen können.

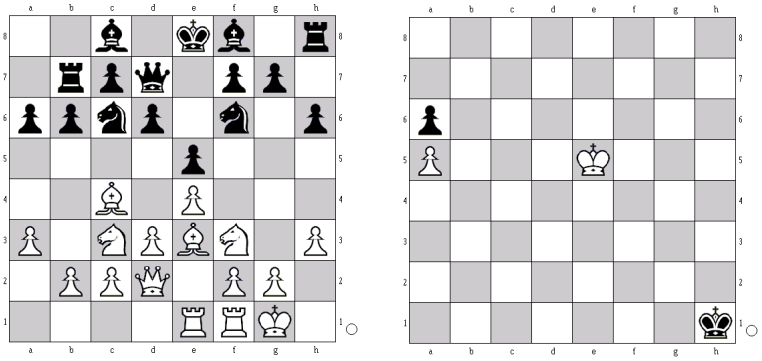

Abbildung 3.1: König in Eröffnung und Endspiel

In der linken Stellung hat es Weiß geschafft, seinen König in Sicherheit zu bringen und kann nun beginnen das Zentrum zu öffnen und mit den gut entwickelten Figuren das Mittelspiel zu starten. Der schwarze Spieler hat dagegen wenig Züge zur Sicherung seines Königs und zur Entwicklung der Figuren unternommen. Im rechten Beispiel, steht der weiße König im Zentrum aktiv und gewinnt die Partie leicht. Der schwarze König steht am Brettrand sehr schwach und kann seinem Bauern nicht mehr helfen.

Aber mit zunehmender Erfahrung und Spielpraxis erlernt der Amateur schnell weitere Kriterien zur Unterscheidung von Stellungstypen für seine Bewertung. Beispielsweise, dass entgegengesetzte Rochadestellungen in den meisten Partien einen großen taktischen Angriff nach sich ziehen können. Geschlossene Stellungen, eher strategisches Lavieren und langfristige Pläne erfordern. In der Schachprogrammierung wird aber genau diesen wichtigen schachtheoretischen Elementen zu wenig Beachtung geschenkt. Ein Ziel des *FUSc#*-Teams[1] ist es den „*Plan im Schach*“ zu kodieren und damit den Schachmotor abhängig von

[1] weitere Informationen unter: *http://www.inf.fu-berlin.de/~fusch*

bestimmten Stellungskriterien nach diesem Plan spielen zu lassen. Es gibt Teilziele, wie z.B. den guten Läufer des Gegners abzutauschen, wichtige Felder zu beherrschen aber auch langfristige, wie einen gezielten Angriff am Königsflügel.

3.2 Schachprogrammierung: Eröffnung, Mittelspiel, Endspiel

Seit Beginn der Schachprogrammierung gibt es sehr wenige *open source*[2] Schachprogramme, die außer den Standardstellungstypen Eröffnung, Mittelspiel und Endspiel noch weitere benutzen. Grundlegend verfolgt man damit die Idee, beispielsweise die Königssicherheit in der Eröffnung sehr hoch zu bewerten und die zentralen Felder als Zielfelder sehr schlecht, damit der König zu Beginn der Partie auf eine Seite in Sicherheit gebracht wird und das Mittelspiel beginnen kann. Im Endspiel dagegen ist der König eine sehr starke Figur und Ziel ist es, den eigenen König in Richtung Brettmitte zu treiben. Mit den Figuren ist es ähnlich. Zu Beginn steht die Entwicklung im Vordergrund und im Mittelspiel das Zentrum und der Vormarsch gegen den gegnerischen König.

Die mit TD-Leaf(λ) verwendete Methode der Koeffizientenoptimierung funktionierte dahingehend sehr gut, dass jeder Stellungstyp relativ gut in einer Partie vertreten ist. Mangelhaft ist jedoch, dass im Schnitt eine Partie einen größeren Mittelspielanteil besitzt, diese Koeffizienten sehr gut gelernt wurden, aber andere, wie z.B. die Endspielkoeffizienten weniger.

Der Ansatz bei *FUSc#* soll nun in Zukunft sein, anhand einer Großmeister-Datenbank eine Stellungsklassifizierung mittels einiger wichtiger Stellungseigenschaften vorzunehmen und diese Stellungstypen dann mit eigenen Koeffizienten zu bewerten. Das Lernverfahren TD-Leaf(λ) kann aber nicht benutzt werden, da einige Stellungstypen häufiger in Partien auftreten als andere. Das würde bedeuten, dass nach z.B. 1000 Partien der Stellungstyp *x* 400-mal optimiert wurde, aber der Stellungstyp *y* erst 5-mal. Wie das Verfahren angeglichen werden muss wird in **Kapitel 4** besprochen.

3.3 FUSc#-Stellungstypen

Die Wahl der 33 Stellungstypen in *FUSc#* wurde mit den zwei starken Turnierspielern Andreas Gropp (2225 Elo) und Christian Düster (2196 Elo) ausgearbeitet. Momentan werden in *FUSc#* alle 32 möglichen Kombinationen aus folgenden Kriterien als Stellungstypen verwendet: beide Damen (ja/nein), König (links, mitte, rechts, zentrum).

Zusätzlich gibt es einen Stellungsvektor `Endspiel`, der dann Gültigkeit besitzt, wenn keine Damen auf dem Brett sind und die Summe aller Läufer, Springer und Türme kleiner 6 oder zwar Damen auf dem Brett sind, die Summe aller Läufer, Springer und Türme kleiner 3 ist.

```
bool endspielkriterium =    (QueenCount==0)&&(LeichteCount<=6)
                         || (QueenCount>=1)&&(LeichteCount<=3);
```

Es genügen wenige Abfragen, um zu entscheiden, welcher Evaluationsvektor durch den entsprechenden Stellungstyp aktiviert werden soll. Im Laufe einer Partie gibt es auch bestimmte Übergänge, die betrachtet werden können, beispielsweise wenn nach 25 Zügen keine Damen mehr auf dem Brett stehen, ist der Pool für mögliche Stellungstypen sehr viel kleiner.

Demnach unterscheidet *FUSc#* mit diesen 33 Vektoren und je 1706 Stellungsfaktoren 56298 Stellungsmerkmale. *KnightCap* verwendete zum Zeitpunkt der Lernexperimente 4 Stellungstypen à 1468 Stellungsfaktoren und unterschied damit 5872 Stellungsmerkmale. Die Überwachung der Stellungsbewertung in *FUSc#* kann mit der GUI-Klasse *Visual Eval 1.0* (siehe Abbildung 3.2) vorgenommen werden.

[2] Mit *open source* werden Programme versehen, deren Quellcode frei verfügbar ist. Der Programmcode des *FUSc#*-Projektes ist ebenfalls einsehbar.

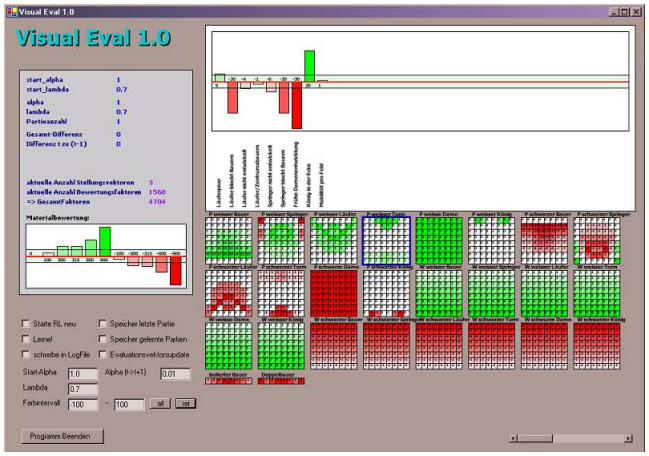

Abbildung 3.2: Visual Eval 1.0

Die Methode **`int Klassifikator(Stellung)`** entscheidet für eine konkrete Stellung, zu welchem Stellungstyp diese gehört und gibt einen fest definierten Index zurück. Dieser Index wird global gespeichert, so kann nun bei Aufruf der Evaluationsmethode mit Eingabe der aktuellen Stellung eine Bewertung vorgenommen werden.

```
int Klassifikator(Stellung s) {
   // Figuren werden gezählt, Positionen ermittelt
   ...
   // Entscheidung -> Stellungstyp
   if (endspielkriterium) {
      // ENDSPIEL!!!
      return 0;
   }
   else {
      // MITTELSPIEL!!!
      if (keine Damen auf dem Brett) {
         // OHNE DAMEN
         if (Weisser König am Damenflügel) {
            // Weiss grosse Seite
            if (Schwarzer König am Damenflügel) {
               // Schwarz grosse Seite; Mittelspiel ohne Damen
               return 1;
            }
            else if (Schwarzer König in der Mitte) {
               // Schwarz in der Mitte; Mittelspiel ohne Damen
         ...
   }
}
```

`evaluationLIST[Vektorindex, Stellungsfaktor]` stellt eine große Matrix dar, deren Spalten die Vektoren repräsentieren. Die Funktion `copyEvaluate2Vector()` kopiert die entsprechend per Hand gesetzten Init-Werte der Evaluation in die Vektoren. Anschliessend haben alle Spalten der `evaluationLIST`-Matrix an den gleichen Indizes die selben Werte.

```
public void copyEvaluate2Vektor() {
```

```
        // für alle Stellungstypen:
        for (int k=0; k<EVALMAXINDEX; k++) {
           // Material:
           // 0 - 6
           for (int i=0; i<=6; i++)
              evaluationLIST[k,i] = e_piece_value[i];
           // Bauerneigenschaft isoliert:
           // 7 - 14
           for (int i=7; i<=14; i++)
              evaluationLIST[k,i] = e_isolated[i-7];
           // Bauerneigenschaft istDoppelbauer:
           // 15 - 22
           for (int i=15; i<=22; i++)
              evaluationLIST[k,i] = e_doublepawn[i-15];
           ...
           // Figurenpositionen (weiss):
           // 32 - 95 BAUER
           // 96 - 159 SPRINGER
           // 160 - 223 LÄUFER
           // 224 - 287 TURM
           // 288 - 351 DAME
           // 352 - 415 KÖNIG
           for (int j=1; j<=6; j++)
              for (int i=32; i<=95; i++)
                 evaluationLIST[k,i+((j-1)*64)] = e_PositionWhite[j][i-32];
           ...
        }
     }
```

Um aber die Evaluationsfunktion auf den Vektor anzupassen, musste diese umgeschrieben werden.
Vorher:

```
evalwert += e_isolated[e_square%8];
```

Nachher:

```
evalwert += evaluationLIST[Stellungsklasse, 7 + e_square%8];
```

Problematisch dabei ist, dass beispielsweise ein Vorzeichenfehler oder sogar eine falsche Indizierung zu einem großen spielerischen Qualitätseinbruch führen kann. Um das zu vermeiden wurde eine Übersetzungstabelle erstellt.

Kapitel 4

TD-Leaf(λ) wird zu TD-Leaf-ComplexEval(λ)

Bei der Verwendung einer sehr umfangreichen Evaluation, die sich auf verschiedenste Stellungstypen bezieht und die Evaluationsfaktoren in diesen Stellungen unterschiedlich bewertet, wäre es ein Problem, eine Lernrate α für das TD-Lernverfahren zu verwenden, da einige Stellungstypen besser als andere gelernt würden. Der Grund dafür ist, dass es Stellungstypen gibt, die seltener auftreten als andere. Bei *KnightCap* gab es auch verschiedene Stellungstypen: Eröffnung, Mittelspiel und Endspiel[1]. Da in den meisten Partien mehrere Züge in diese Klassen fallen, werden sie ungefähr gleich oft angepasst. Der Schachmotor *FUSc#* besitzt nun eine sehr viel komplexere Evaluationsprozedur, dass es sehr wahrscheinlich ist, so dass nach dem Lernen einige der Stellungsvektoren kaum oder sehr schlecht angepasst werden. Um nun dieses Problem zu beheben, werden für jeden Stellungsvektor $s_1, s_2, ..., s_k$ entsprechend unterschiedliche Lernfaktoren $\alpha_1, \alpha_2, ... \alpha_k$ eingeführt die nach festgelegten Regeln individuell angepasst werden.

Die Aktualisierungsformel des 56298-elementigen Koeffizientenvektors ω, der aus den aneinander gehängten, partiellen Stellungsvektoren $\omega_1, \omega_2, ..., \omega_{33}$ besteht, muss nun dahingend verändert werden, dass die $\alpha_1, \alpha_2, ..., \alpha_{33}$ den entsprechenden Stellungsvektoren zugeordnet werden können.

$$\omega_k := \omega_k + \alpha_k \sum_{t=1}^{N-1} \nabla J'_d(x_t, \omega_k) \left[\sum_{j=t}^{N-1} \lambda^{j-t} d_j \right],$$

, wobei $k = 1, 2, ..., 33$ die entsprechenden Stellungsklasse zu der verwendeten Stellung x_t ist.

4.1 TD-Leaf-ComplexEval(λ)

Im folgenden, der Pseudocode des in *FUSc#* implementierten Algorithmus zur Berechnung der Temporalen Differenz-Methode in Kombination zu einem Suchalgorithmus. Der Unterschied zu dem in *KnightCap* verwendeten TD-Leaf(λ)-Algorithmus ist zum einen die Verwendung der in einer Partie berechneten Daten und zum anderen die unabhängige Optimierung der jeweils in einer Partie auftretenden Stellungstypen.

```
Sei J(.,w) eine Klasse von Evaluierungsfunktionen, parametrisiert
mit ω ∈ ℝ^k. Seien weiterhin x_1,...,x_N  N eigene Stellungen, die während
einer Partie betrachtet werden und r(x_N) das Resultat.
Als Konvention gelte: J(x_N, ω) := r(x_N).
Vorbereitung in einer Partie:
   p(x_i), letzte Stellung der Hauptvariante (Länge p) von
   Stellung x_i mit x_{i_1}, x_{i_2}, ..., x_{i_p}
Lernschritt:
```

[1] Die meisten Schachprogramme verwenden diese Klassifikation.

1. for j=x_N to x_1 do begin
   1.1 gehe zu Stellung $p(x_j)$
   1.2 berechne den Gradienten von $J(p(x_j),\bullet)$
   1.3 berechne die Temporalen Differenzen
   $$d_t := J(p(x_{t+1}),\omega) - J(p(x_t),\omega)$$
   (mit Ausnahme von d_N)
   1.4. gehe zu Stellung x_j zurück
end
2. Update ω anhand der TD-Leaf(λ)-Formel:

$$\omega_k := \omega_k + \alpha_k \sum_{t=1}^{N-1} \nabla J(p(x_t),\omega_k) * \left[\sum_{j=t}^{N-1} \lambda^{j-t} d_j\right]$$

4.2 TD-Leaf-ComplexEval(λ) in FUSc#

Die Implementierung von TD-Leaf(λ) in *FUSc#* unterscheidet sich sehr von der in *KnightCap*. Eine Idee zur Verbesserung besteht z.B. darin, Informationen, die in einer Partie gesammelt werden, wie z.B. die Hauptvariante und die entsprechend erhaltenen Stellungswerte, in einer Datei zu speichern und anschließend beim Lernvorgang diese zu verwenden.

4.2.1 Vorbereitungs- und Spielphase

Um nicht nach jeder gespielten Partie die Daten verändern zu müssen, sollte ein Automatismus dieses übernehmen. Eine exemplarische Partie wurde mit *FUSc#* gespielt und dabei wurden folgende Werte in die Datei *rllastgame.rl* nach folgendem Format gespeichert:

```
// Anzahl der bisher gelernten Partien
0
// nun folgen die jeweiligen Alpha-Werte 1-33
1
1
...
1
// der verwendete Lambda-Wert
0,7
// die verwendeten Koeffizienten der Stellungsvektoren (>51000)
0 200 620 650 1000 1800 20000 -2 -3 -4 -5 -5 -4 -3 -2 -10 -6 -8 -10
-10 -8 -6 -10 5 -20 0 -1 0 -20 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 3 2 6 6 6 6 2 3 3 3 8 9 9 6 3 3 5 5 9 11 11 9 5 5 9 9 11 13 12 11
9 9 13 13 14 15 15 14 13 13 0 0 0 0 0 0 0 0 -5 0 0 0 0 0 0 -5 -1 0
2 3 3 2 0 -1 -3 1 4 2 2 4 1 -3 -2 1 3 3 3 3 1 -2 0 2 2 2 2 2 2 0 0
1 1 1 1 1 1 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 2 2 4 0
3 3 0 4 2 0 2 4 2 2 4 2 0 0 2 2 4 4 2 2 0 0 2 2 2 2 2 2 0 0 0 2 2 2
2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 4 5 5 4 1 0 0 0 0 4 4 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1 2 2 1 1 1 1 1 1 2 2 1 1 1 1 1 3 4 4 3 1 1 1 3 5 7 7 5 3 1 3 5 7
...
-2 -2 -2 -2 -2 -2 -2 -2 -1 -1 -1 -1 -1 -1 -1 -1 0 -2 2 3 4 5 5 4 3 2
// Spielfarbe von FUSc#
weiss
// Anzahl der Halbzüge in der Partie
101
// Zug Bewertung Zeit-Weiss Zeit-Schwarz Hauptvariante (eigene Informationen)
```

```
e2e3 18 300000 300000 e2e3 c7c5 c2c3 d7d5 f2f4 b7b6
// Vom Gegner erhalten wir nur die Information des gespielten Zuges
d7d5 0 0 0 leer
c2c3 18 291723 293762 c2c3 c7c5 g1e2 a7a6 f2f3
e7e5 0 0 0 leer
g1e2 18 286891 290578 g1e2 b8c6 d1a4 g7g5 d2d3
c7c5 0 0 0 leer
d1c2 17 279435 288595 d1c2 b8c6 f2f3 d5d4 e3e4
b8c6 0 0 0 leer
f2f3 17 268545 287373 f2f3 d5d4 e3e4
g8f6 0 0 0 leer
a2a4 21 262181 282976 a2a4 e5e4 f3f4
g7g6 0 0 0 leer
g2g4 24 252673 280702 g2g4 e5e4 h2h4
f8g7 0 0 0 leer
h2h3 29 229715 278219 h2h3 e5e4 g4g5
c8e6 0 0 0 leer
b2b3 30 225754 271049 b2b3 c5c4 b3b4 g6g5
d8c7 0 0 0 leer
g4g5 37 218219 269777 g4g5 f6h5 d2d3
f6h5 0 0 0 leer
b1a3 31 202421 267784 b1a3 d5d4 a3b5
a7a6 0 0 0 leer
d2d3 37 183118 266422 d2d3 d5d4
d5d4 0 0 0 leer
e3e4 29 180880 263318 e3e4 d4c3 e2c3 a6a5
d4c3 0 0 0 leer
c2c3 21 173043 262527 c2c3 f7f6 h3h4 f6g5
b7b6 0 0 0 leer
h3h4 29 170895 261345 h3h4 a6a5
...
e3d2 0 0 0 leer
g2g1 219 53638 78369 g2g1 f8c5 g1h1 d2c1 f6c6
// Partieergebis (1 Sieg, 0 Remis, -1 Niederlage)
// Dieser Wert wird aber erst direkt vor dem Lernvorgang ermittelt, da UCI es nicht vorsieht,
// dem Schachmotor das Resultat mitzuteilen.
0
// Der entsprechene Reward-Wert wird anschliessend gesetzt
0
```

Weder die Beendigung einer Partie, noch das Resultat wird unter dem UCI-Protokoll dem Schachmotor mitgeteilt. Er kann dies lediglich indirekt über den Beginn einer neuen Partie feststellen. Eine Partie wird gespielt, indem der Schachmotor eine bestimmte Zugfolge erhält und anschließend aufgefordert wird, einen Folge-Zug zu liefern. Nun kann man die bisher gespielten Züge speichern und damit feststellen, ob eventuell eine neue Partie beginnt, oder die alte weitergeführt wird.

```
// altgame[] enthält die bisher gespielte Partie
// values[] enthält die aktuell übergebene Zugfolge
bool inPartie=true;
for (int i=0; i<values.Length-1; i++)
   if (values[i]!=altgame[i])
      inPartie = false;
```

Nach Berechnung und Rückgabe eines Zuges vom Schachmotor, wird dieser in der Datei *rllastgame.rl* mitprotokolliert.

```
// Wenn FUSc# im Lernmodus ist, soll der letzte Zug gespeichert werden
if (ReinforcementLearning.SAVELASTGAME)
{
   ReinforcementLearning.SetNextMove(MoveToString(aktBestMove),
                                     aktBestScore,
                                     a_WhiteTimeNow,
                                     a_BlackTimeNow,
                                     aktBestPath);
}
```

Wurde nun eine neue Partie begonnen, wird zunächst überprüft, ob die gespeicherte Datei *rllastgame.rl* eine gültige Partie mit einem eindeutigen Resultat enthält. Wenn das der Fall ist, kann der Lernalgorithmus gestartet werden. Als Eingabeparameter benötigt dieser:

```
double[] ALPHA ...Lernraten
double LAMBDA ...Einflussfaktor (Default ist 0.7)
int stk ...Anzahl verschiedener Stellungsvektoren(Evaluation)
int efk ...Anzahl verschiedener Stellungsfaktoren je Vektor
int N ...Anzahl Züge der zu lernenden Partie(schwarz und weiss)
Move[] game[1..N] ...letzte gespielte Partie (zu lernen)
Path[] path[1..N] ...gespeicherte Hauptvarianten während einer Partie; Länge kann variieren
int result ...1 für Sieg, 0 für Remis und -1 für Niederlage (aus Sicht von Fusc#)
Color fuschColor ...Spielfarbe von FUSc#
```

Nun wird das aktuelle Board auf die Startstellung initialisiert.

```
Game.Manager.NewBoard.ResetBoard();
```

Anschliessend wird die Partie einmal auf dem Brett durchlaufen, bis zu der Stellung, mit dem letzten gespielten FUSc#-Zug.

```
for (i=0; i<ReinforcementLearning.MoveListLength; i++)
   MoveForward(ReinforcementLearning.moveList[i]);
```

Um die Optimierungsformel

$$\omega_k := \omega_k + \alpha_k \sum_{t=1}^{N-1} \nabla J'_d(x_t, \omega_k) \left[\sum_{j=t}^{N-1} \lambda^{j-t} d_j \right]$$

des Koeffizientenvektors berechnen zu können, müssen zunächst alle Temporalen Differenzen berechnet werden. Die in der Partie ermittelten Stellungswerte werden in dem `evalint[]`-Array gespeichert. Der letzte Index wird mit dem Reward der Partie bewertet.

```
evalint[N-1] = getReward(RESULT, ReinforcementLearning.color);
```

Nun wird in einer Schleife die Partie Zug für Zug rückwärts gespielt und dabei nur jede zweite Stellung betrachtet. Für die Verwendung von der Temporalen Differenz-Methode sind nur die eigenen Stellungen und Aktionen relevant.

```
for (i=N-2; i>=0; i--)
{
   MoveBackward();
   if (ReinforcementLearning.color)
   {
```

```
        evalint[i] = ReinforcementLearning.moveScore[i*2];
        HV[i] = ReinforcementLearning.HVList[i*2];
    }
    else if (!ReinforcementLearning.color)
    {
        evalint[i] = ReinforcementLearning.moveScore[i*2+1];
        HV[i] = ReinforcementLearning.HVList[i*2+1];
    }
```

Zu jeder Stellung wurde die Hauptvariante gespeichert, diese liegt nun in HV[i] vor. Die Idee von TD-Leaf ist es, diese Hauptvariante abzulaufen und damit eine forcierte gute Stellung zu evaluieren.

```
    string[] pfad = this.splittingString(HV[i]);
    while (way<pfad.Length)
    {
        MoveForward(pfad[way]);
        way++;
    }
```

Die aktuelle Bewertung der Stellung und die der nachfolgenden sind vorhanden, `td[i]` speichert die Temporale Differenz von x_{t+1} und x_t

$$d_t := J'_d(x_{t+1}, \omega) - J'_d(x_t, \omega)$$

```
    td[i]      = evalint[i+1] - evalint[i];
```

Ist die Temporale Differenz positiv, so hat unser Gegner möglicherweise einen Fehler gemacht, diesen wollen wir nicht mitlernen. Demzufolge setzen wir diesen dann auf 0.

```
    if (td[i]>0)
        td[i]=0;
```

Es sei denn, dem Gegner blieb nicht anderes übrig und wir hätten in dieser Stellung den gleichen Zug gespielt.

Im nun folgenden Abschnitt werden die partiellen Ableitungen der Stellungsfaktoren `efk` und Stellungstypen `stk` berechnet und für jede Stellung i im Array `nabla[i,stk,efk]` gespeichert. Zunächst wird der in der Partie verwendete Vektor in `saveVektor` gespeichert und anschliessend jeder Faktor mit 0 initialisiert.

```
saveVektor = (int[,])evaluationLIST.Clone();
for (a=0; a<stk; a++)
    for (b=0; b<efk; b++)
        evaluationLIST[a, b] = 0;
```

Normalerweise sollte `delta` sehr klein sein. Da die FUSc#-Evaluation aber auf einer reinen Integer-Berechnung basiert, ist der kleinste Wert, zur Veränderung eines Faktors gerade 1.

```
int delta = 1;
```

Es ist nicht notwendig, alle Stellungstypen zu betrachten.

```
// bevor wir starten, ermitteln wir den Stellungsvektor, um nur dort die
// Parameter zu lernen
aktStellungsTyp = Game.Manager.NewBoard.Klassifikator();
```

Der aktuelle Stellungstyp wurde durch den Klassifikator identifiziert und der Index in `aktStellungsTyp` gespeichert. Folgende einfache Möglichkeit, diese partielle Ableitung zu berechnen, wurde implementiert [BaxTriWea2]:

$$\nabla J'_d(x_t, \omega_k) = (\frac{\partial J'_d(x_t,\omega_k)}{\partial \omega_{k_1}}, \frac{\partial J'_d(x_t,\omega_k)}{\partial \omega_{k_2}}, ..., \frac{\partial J'_d(x_t,\omega_k)}{\partial \omega_{k_j}})$$

und für den einen Faktor *y* gilt:

$$\frac{\partial J'_d(x_t,\omega_k)}{\partial \omega_{k_y}} \approx \frac{J'_d(x_t,\omega_{k_y}+\delta) - J'_d(x_t,\omega_{k_y})}{\delta}$$

da die kleinste Veränderungseinheit 1 ist (*FUSc#* arbeitet nur mit ganzzahligen Werten, da die Berechnung von Integerwerten auf dem Prozessor schneller geht) vereinfacht sich die Berechnung zu

$$\frac{\partial J'_d(x_t,\omega_k)}{\partial \omega_{k_y}} \approx J'_d(x_t, \omega_{k_y} + 1) - J'_d(x_t, \omega_{k_y})$$

, wobei *k* Stellungsklasse und *j* Gewicht eines Stellungsfaktors ist.

```
for (b=0; b<efk; b++)
{
   evaluationLIST[aktStellungsTyp, b] = saveVektor[aktStellungsTyp, b] + delta;
   update = RuheSuche(-MATT, MATT, (GetWhiteToMove()==1), 0);
   evaluationLIST[aktStellungsTyp, b] = saveVektor[aktStellungsTyp, b];
   update -= RuheSuche(-MATT, MATT, (GetWhiteToMove()==1), 0);
   nabla[i, aktStellungsTyp, b] += update
   evaluationLIST[aktStellungsTyp, b] = 0;
}
```

Der Hauptvariantenpfad wird wieder zurückgelaufen, damit die nächste Stellung untersucht werden kann.

```
   way--;
   while (way>=0)
   {
      Game.Manager.NewBoard.MoveBackward();
      way--;
   }

   // Falls "am ersten Zug" angekommen, so nimm den Zug nicht mehr zurück
   // Ansonsten überspringe die schwarzen (bzw. weissen) Züge
   if (ReinforcementLearning.color)
   {
      if (i>0)
         Game.Manager.NewBoard.MoveBackward();
   }
   else if (!ReinforcementLearning.color)
   {
      if (i>1)
         Game.Manager.NewBoard.MoveBackward();
   }
} // ENDE: for (i=N-2; i>=0; i--)
```

4.3 Versuche und Ergebnisse

Um einmal die Wirkung eines Lernschrittes mit dem TD-Leaf-ComplexEval(λ) zu verdeutlichen, sollen die folgende zwei Blitzpartien helfen (keine Eröffnungsbücher, beide Schachmotoren spielen deterministisch). Partie 1 wurde gegen den Schachmotor *TRex*[2] gespielt und im 16. Zug verloren. Jedem Motor standen 3 Minuten für die komplette Partie zur Verfügung. Anzumerken ist, dass *TRex* kein sehr starkes Programm ist, welches die Dame in das Mobilitätskriterium aufgenommen hat und daher die Dame zu früh entwickelt.

FUSc# - TRex

1.d4 - e6, **2.**e4 - Sc6, **3.**Sf3 - Df6, **4.**Lg5 - Dg6, **5.**Sc3 - Lb4, **6.**e5 - De4+, **7.**Le2 - Dg4,

8.h3 - De4, **9.**0-0 - Dg6, **10.**Ld3 - Dh5, **11**.a3 - h6, **12.**axb - hxg, **13.**b5 - Sb4,

14. Le4 - f5, **15.**g4 - Dxh3, **16.**gxf - Dh1# (Matt)[3]

Nachdem *FUSc#* die Partie benutzt hat, um die Evaluation anzupassen (es wurden in 2 Stellungstypen insgesamt 35 Parameter beim Lernvorgang verändert), lief die zweite Partie mit den optimierten Koeffizienten bis zum 13.Zug identisch (siehe Abbildung 4.2).

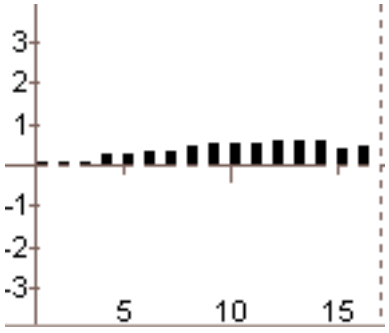

Abbildung 4.1: Bewertungskurve von *FUSc#* der Partie: *FUSc# - TRex* (Partie 1)

Bis dahin hatte die Bewertungskurve (Abbildung 4.1) im Laufe der ersten Partie einen stetigen Anstieg zu verzeichnen.

Jetzt wich *FUSc#* mit der folgenden Zugfolge ab und gewann die zweite Partie im 51.Zug.

14. Dd2 - g4, **15.**Sg5 - Sxd3, **16.**cxd - gxh, **17.**gxh - f6, **18.**exf - gxf, **19.**Se4 - Dxh3, **20.**f4 - b6,

21. Kf2 - d5, **22.**Th1 - Dxh1, **23.**Txh1 - Txh1, **24.**Kg2 - Ta1, **25.**Sg3 - ...

Die Bewertungskurve der zweiten Partie (Abbildung 4.3) zeigt eine Veränderung der Bewertung. *FUSc#* plant einen anderen Weg zu gehen und sieht sich mit diesem (in dem kleinen Horizont von 5 Halbzügen) als besser, unterschätzt aber wieder die am Königsflügel entstehenden Mattdrohungen und verteidigt sich eifrig. Der Einbruch der Bewertung im 21. Zug ist damit zu erklären, dass *TRex* den gut postierten und wichtigen Springer angreift, dem nun wenig Felder zur Verfügung stehen. *TRex* entscheidet sich nach **22.**Th1-... die Dame für zwei Türme zu geben und verliert durch die schlechte Bauernstruktur und Königssicherheit.

Die taktischen Spielqualitäten beider Schachmotoren in den Partien sind nicht mit professionellen Programmen vergleichbar, so haben beide in den Blitzpartien nur eine durchschnittliche Suchtiefe von $5-6$ Halbzügen erreicht, doch wird durch diese beiden Partien der Einfluss des Lernverfahrens gut ersichtlich.

[2] Der Autor ist *Christophe Drieu* (Frankreich). Der Schachmotor kann unter h*ttp://membres.lycos.fr/refigny63/trex.zip* geladen werden.

[3] *FUSc#* übersah das Matt, da im zur Suche wenige Sekunden zur Verfügung standen und ein Matt als solches im Suchbaum (bei *FUSc#*) erst zwei Halbzüge später gefunden wird.

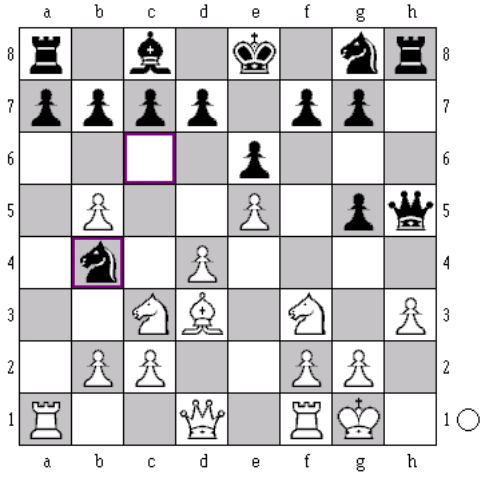

Abbildung 4.2: *FUSc#* - *TRex* (Partie 1 nach **13.** ...-Sb4)

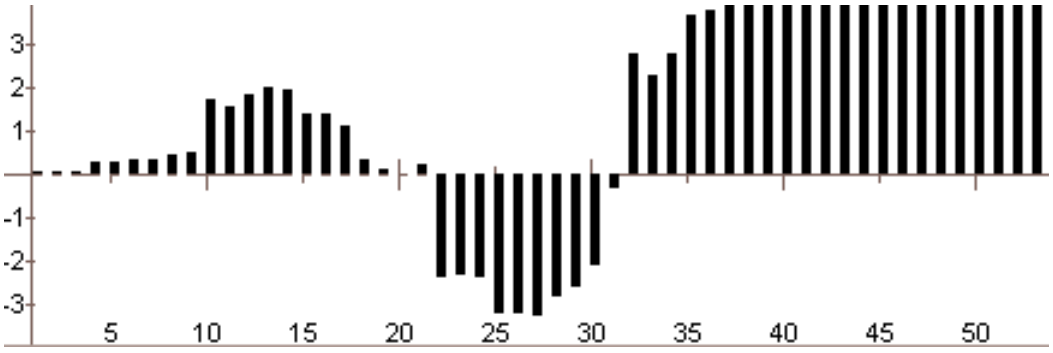

Abbildung 4.3: Bewertungskurve von *FUSc#* der Partie: *FUSc#* - *TRex* (Partie 2)

FUSc# auf dem Schachserver *http://www.schach.de*

Um die aktuelle Spielstärke zu ermitteln, spielte *FUSc#* zunächst 50 Partien gegen menschliche Spieler in verschiedenen Spielmodi ($1-5$ Minuten Bedenkzeit). Die Bewertung auf diesem Schachserver ergab 1800 ± 50. Verwendet wurde ein per Hand eingestellter Vektor, der jedem Stellungstyp zugeordnet wurde. *FUSc#* unterschied demnach keine der Stellungstypen und behandelte alle gleich. Startwert für die Lernraten $\alpha_1, \alpha_2, ..., \alpha_{33}$ war 1.0 und für $\lambda = 0.7$. Schon nach wenigen Partien war eine leichte Steigung der Performance zu verzeichnen, so nahm *FUSc#* in der Bewertung Änderungen an den Figurenpositionen und deren Wirkung vor.

Ein erstes Problem bei der Wahl der Stellungstypen zeigte sich, so werden wenige Stellungstypen sehr oft und andere fast garnicht gespielt und gelernt. Nach 7 Partien ergab sich folgende Verteilung (siehe Abbildung 4.4) auf die 33 Stellungstypen.

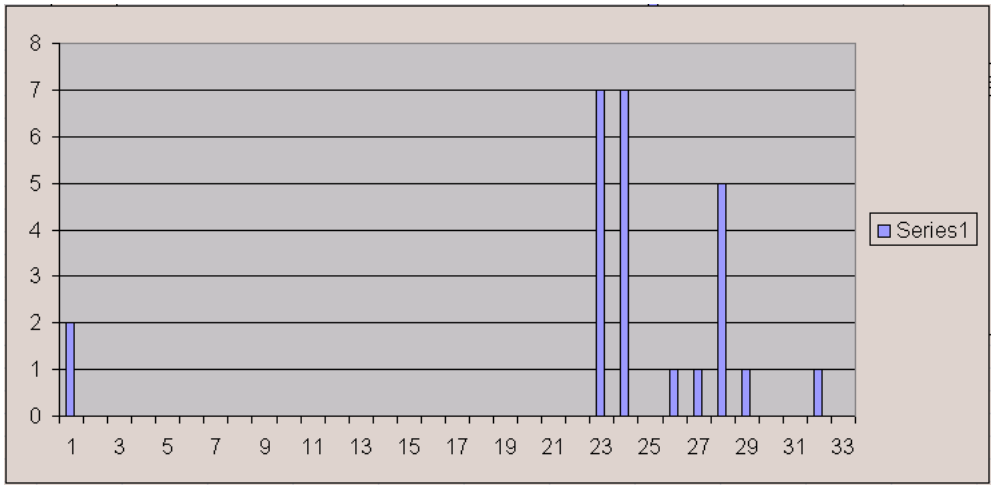

Abbildung 4.4: Verteilung der Stellungstypen nach 7 Partien

Die Abbildung zeigt die Lernraten zu den 33 Stellungstypen nach 7 gespielten Partien. Initialisiert wurden die Lernraten mit 1.0. Nach jedem Lernschritt werden die Lernraten verkleinert. Stellungsmuster 23 und 24 traten am häufigsten auf.

Die beiden Stellungstypen 23 und 24 traten am häufigsten auf. Stellungstyp 23 repräsentiert die Eröffnungsphase, beide Könige stehen auf der Mitte der Grundreihe. Im Stellungstyp 24 hat schwarz und bei 28 haben beide klein rochiert.

Zum Zeitpunkt der Abgabe dieser Diplomarbeit, hatte *FUSc#* 119 Blitz-Partien gespielt und seine Performance von 1800 ± 50 auf 2016 (Abbildung 4.5) erhöht.

Von den 119 Partien wurden 72 verwendet, um die Koeffizienten anzupassen. Es gab Partieentscheidungen durch Verbindungsabbrüche und Abstürze des Schachmotors, diese Partien wurden nicht zum Lernen verwendet. Die Verteilung der aufgetretenen Stellungstypen zeigt Abbildung 4.6.

Eine Verbesserung der Spielstärke war schon nach einigen Partien erkennbar, doch benötigt *FUSc#*, um eine Stellung gut zu kennen und diese optimal bewerten zu können, für den Umfang der insgesamt über 56000 Faktoren in den 33 Stellungstypen bei einem gleichverteiltem Auftreten dieser, schätzungsweise weit über 50000 Trainingspartien. Dieser Testlauf wird leider erst nach Abgabe dieser Diplomarbeit beendet werden können. Es lässt sich aber anhand der erhaltenen Ergebnisse sagen, dass die Optimierung der per Hand eingesetzten Werte (jede Stellungstyp wurde gleich behandelt) die Spielqualität wesentlich erhöht hat. Nach 72 gelernten Partien hat sich die Leistung von 1800 auf über 2000 erhöht. Ein wichtiger Faktor dabei war die Königssicherheit, so hatte FUSc# gelernt, dass der König in einigen Stellungstypen aktiviert werden muss und im Zentrum besser am Spiel teilnehmen kann.

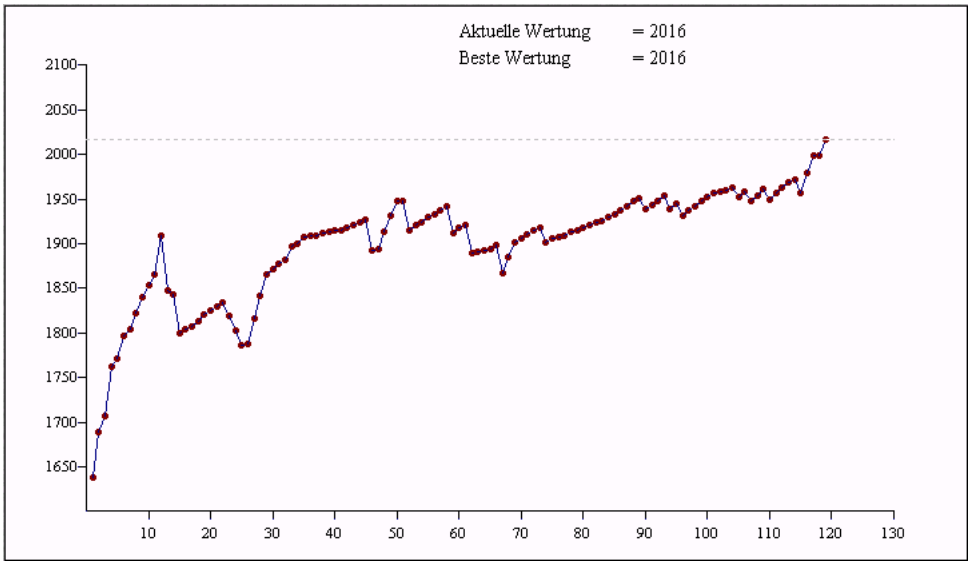

Abbildung 4.5: Bewertung auf dem Schachserver nach 119 Blitz-Partien

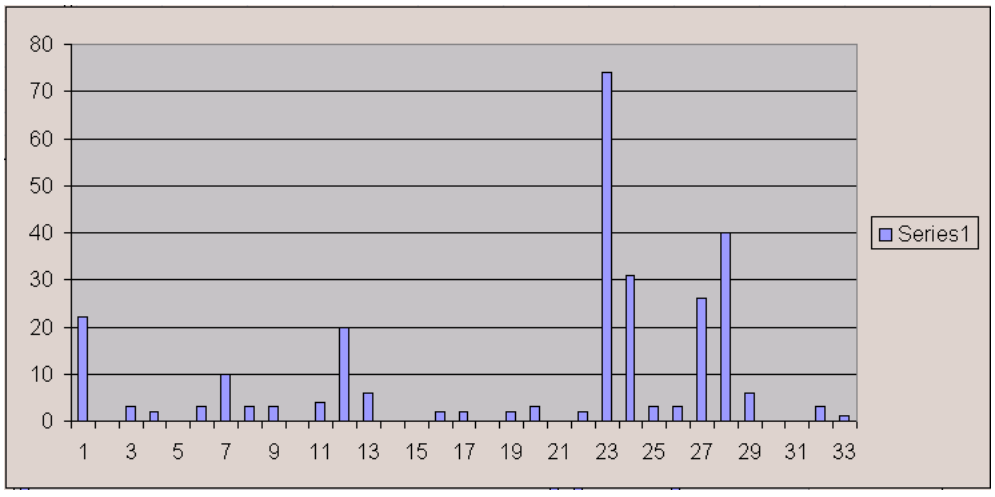

Abbildung 4.6: Verteilung der Stellungstypen nach 72 gelernten Partien

Die Abbildung zeigt die Verteilung der 33 Stellungstypen in den 72 gelernten Partien. Der Stellungstyp 23 kam in jeder Partie vor, da er die Eröffnung repräsentiert.

Kapitel 5

Diskussion und Ausblick

5.1 Diskussion

Der positive Einsatz von TD-Leaf(λ) im Schachmotor *KnightCap* hat sich auch bei *FUSc#* bestätigt. Da die Evaluationfunktion mit den zahlreich verwendeten Stellungstypen sehr umfangreich ist, benötigt *FUSc#* eine sehr viel größere Anzahl an Partien, um die Koeffizienten auf eine annehmbares Niveau zu bringen, als es bei *KnightCap* der Fall war. Es ist bei einem menschlichen Spieler auch nicht zu erwarten, dass er „über Nacht“ zum Großmeister wird. Im übrigen sind die Lernerfolge beider Schachmotoren sehr schwer zu vergleichen, da *KnightCap* wesentlich tiefer rechnet und daher diese taktische Komponente den Ausschlag gibt. In sehr vielen Partien stand *FUSc#* positionell überlegen und entwickelte sich strategisch sehr gut, übersah aber oft taktische Verwicklungen und verlor damit die Partien.

Um den Automatismus beim Lernen besser umsetzen zu können, muss das UCI-Protokoll aber grundlegend geändert werden. Beispielsweise sollte dem Schachmotor am Ende das Resultat einer Partie mitgeteilt werden und es muss ihm durch eine Art Ruhezustand die Möglichkeit gegeben werden, Parameter anzugleichen und somit einen Lernprozess zu starten.

Wahl der Stellungstypen

Es hat sich bei den Spielen auf dem Schachserver gezeigt, dass die Wahl der 33 Stellungsklassen etwas unglücklich war. Das Problem bestand darin, dass es einen grossen Unterschied macht, ob beide Damen vom Brett verschwunden sind, oder nur eine. Unsere Klassifikation unterscheidet das nicht. Es gab viele Stellungsmuster, die nie auftraten.

Der Eröffnungsphase hingegen, stand nur ein Vektor zur Verfügung. Besser wäre es gewesen, dieser Phase mehr Spielraum zum Lernen zu geben.

5.2 zukünftige Arbeiten

Die Wahl der 33 Stellungstypen in *FUSc#* war nicht sehr glücklich. Das Schachprogrammierteam wird in nächster Zeit die Großmeisterdatenbank in Angriff nehmen, um eine bessere Klassifizierung zu finden. Es sollen mit dem Klassifikationsverfahren *Expectation Maximisation* (EM) sehr viele unterschiedliche Stellungstypen identifiziert werden. Die Klassifizierung soll anhand einfacher und komplexer Muster (Königsstellung, Position der Bauern) stattfinden. Parallel wird die Suchtiefe ständig erweitert und die Performance gesteigert. Die Koeffizienten der erhaltenen Stellungstypen werden dann wieder durch zahlreiche Trainingspartien optimiert. Die Arbeit an den 33 bisher verwendeten Stellungstypen hat gezeigt, dass der Trainingsvorgang wesentlich mehr Zeit in Anspruch nimmt und eine Erweiterung der Stellungsklassen auf z.B. 1000 wahrscheinlich hundert-tausende Spiele erfordert. Aber auch ein menschlicher Schachspieler,

benötigt viele Jahre, um es zur Meisterschaft zu bringen. Zusätzlich zu einem harten Training sind aber auch Talent und Kreativität gefragt.

Die zweite große Herausforderung ist der „*Plan im Schach*“. Dazu wird ein abstraktes Planmodell entworfen und Züge im Suchalgorithmus höher bewertet, die zu diesem Plan passen. Beispielsweise erfordert eine Stellung mit gegenseitigen Rochaden, Angriffe an den jeweiligen Flügeln. Züge, die einen Vormarsch der Bauern oder den Einbruch der Figuren am Flügel des gegnerischen Flügels bedeuten, werden besser bewertet. Aber auch einfachere Pläne, wie den Abtausch bestimmter Figuren, um damit in bessere Stellungen zu gelangen, werden abhängig vom Stellungstyp untersucht.

Momentan wird der Einsatz von Neuronalen Netzen zur Zugvorhersage diskutiert. Ein Neuronales Netz mit dem Wissen von einigen millionen Großmeisterpartien kann dem Suchalgorithmus Vorschläge unterbreiten, wie: „... untersuche zunächst mal diesen Zug, in solchen Stellungen wird er häufig gespielt ...“. Ist dieser Zug dann nicht viel schlechter als andere, könnte er strategisch gesehen besser sein.

Die *AG Schachprogrammierung* des Fachbereichs Mathematik/Informatik ist unter folgendem Link zu finden:

http://page.mi.fu-berlin.de/~fusch/

Kapitel 6

Anhang

6.1 Anleitung für die Einbettung von TD-Leaf(λ) in einen Schachmotor

Ein mögliche Vorgehensweise wäre die explizite Verwaltung der Partieergebnisse durch den Programmierer und eine anschließende Anpassung des Vektors. Doch wer möchte schon tagelang seinen Schachmotor überwachen und "per Hand" die Daten aktualisieren. Darum wird im folgenden ein algorithmisches Vorgehen angegeben, was dem Schachprogramm etwas Unabhängigkeit in dieser Hinsicht bringt.

Die folgende kleine Anleitung soll einen Wegweiser geben, um die Bewertungsfaktoren eines bereits vorhandenen Schachmotors mit TD-Leaf(λ) eigenständig lernen zu lassen. Die Einbettungsdauer hängt vom Umfang der Evaluation ab. Je grösser die "schachtheoretischen Matrizen"[1], desto länger (aber effektiver) arbeitet der Algorithmus. Die folgende Vorgehensweise ist zunächst allgemein gefasst und mit *FUSc#*-Code als Beispiel kompletiert.

1. Evaluationsvektor als Vektor repräsentieren und in ein Dateiformat packen

Dazu können die entsprechenden Werte und Listen durch eine zusätzliche Funktion "gefüllt" werden. Es sollte ein Array (z.B. `eval_vector[]`) angelegt und mit den gewünschten Daten versehen werden. Dabei können sich offsets als grosse Hilfe erweisen, da die Wahrscheinlichkeit einen Fehler zu produzieren mit der Anzahl der Bewertungsfaktoren steigt. Diesem Arbeitsschritt muss eigentlich die größte Aufmerksamkeit gewidmet werden, da hier produzierte Fehler mit aller Wahrscheinlichkeit nicht mehr oder kaum noch zu finden sind. Eine Überführungstabelle der Form:

```
...
piece_value[1] = eval_vector[48]
piece_value[2] = eval_vector[49]
...
piece_value[6] = eval_vector[53]
...
```

ist anzuraten.

Vorher:

```
...
private int[] piece_value = {0, 100, 300, 315, 500, 900, 10000};
private int[] isolated    = { -2, -3, -4, -5, -5, -4, -3, -2};
```

[1] Damit seien die Bewertungsmatrizen z.B. für die Figurenpositionen gemeint.

```
private int[] doublepawn  = { -5, -3, -4, -5, -5, -4, -3, -5};
...
```

Nachher:

```
...
private int[] piece_value;
private int[] isolated;
private int[] doublepawn;
...
initializeEvalVector()
{
   for (int i=0; i<6; i++)
      piece_value[i] = eval_vector[i+offset_piece_values]
   ...
}
...
```

2. Bei jedem Programmstart sollten die Bewertungsfaktoren eingelesen werden.

Dazu muss eine Datei angelegt werden, welche den Inhalt des Bewertungsvektors (`eval_vector[]`) speichert. *FUSc#* verwendet ein einfaches Textdateiformat und zur Verwaltung des Vektors drei Funktionen:

`readEvalVectorFromFile()` zum Einlesen der Daten

`copy2Evaluate()` die eingelesenen Daten werden auf die entsprechenden Bewertungsparamter übertragen

`saveEvalVectorToFile()` zum Speichern der Daten

3. Das Programm muss zu erkennen in der Lage sein, wann eine Partie neu beginnt oder weiterläuft.

Zu diesem Zeitpunkt gibt es zwei mögliche Protokolle (WinBoard und UCI) mit Schachoberflächen zu kommunizieren. Wir haben uns für das neue Protokoll UCI (Universal Chess Interface) entschieden. Um dem Schachmotor eine Stellung zu übergeben, wird z.B. eine Zugfolge von der Startposition aus angeben und nach einem weiteren Schlüsselwort, beginnt der Motor mit der Zugberechnung. Zur Verdeutlichung soll folgendes Beispiel dienen. Der Schachmotor beginnt das Spiel in der Startstellung mit Weiß. Nachdem er den besten Zug geliefert hat, ändert sich in einer Partie normalerweise nur der letzte Zug. Demzufolge kann der Schachmotor, beim mitspeichern der bisher gespielten Züge erkennen, ob er sich in der gleichen Partie (mit gleicher Farbe) befindet oder nicht.

Es gibt also zwei Möglichkeiten eine neue Partie zu beginnen, die eine wäre in der Startstellung mit weiss oder schwarz, die andere eine Zugfolge die in keinem Kontext mit der zuvor abgegebenen Stellung steht. Da es für TD-Leaf(λ) aber notwendig ist eine komplette Partie zu betrachten, kommt die zweite nicht in Frage. Ein Wahrheitswert `gueltigePartie=true` kann bei der Startstellung gesetzt werden und invertiert seinen Wert bei Verletzung der Partiehistorie.

UCI-Beispiel für eine gültige Partie:

```
position startpos
go
bestmove e2e4
position startpos moves e2e4 e7e5
go
bestmove g1f3
usw...
```

UCI-Beispiel für eine ungültige Partie:

```
position startpos
go
bestmove e2e4
position startpos moves c2c4 e7e5
go
bestmove g1f3
usw...
```

4. Nach jedem Zug müssen die Hauptvariante, der entsprechende Zug und die Bewertung gespeichert werden.

Die Speicherung der Hauptvariante stellt eine Verbesserung zu der in *KnightCap* vorgestellten Lernvariante dar. Der zeitliche Aufwand, einen guten Weg durch den Suchbaum zu finden, wurde bereits vom Schachmotor in der Partie vorgenommen, dementsprechend können wir auf diese Ergebnisse zurückgreifen und beschleunigen den Lernablauf, sodass dieser vor einer Partie problemlos ausgeführt werden kann. Es scheint auch plausibel zu sein, dem Schachmotor die gleiche Zeit und den gleichen Suchraum im Lernverfahren, wie in der Partie zur Verfügung zu stellen.

Bei dieser Variante ist aber darauf zu achten, dass bei der Verwendung der Ruhesuche nach dem Horizont der Hautpvariante, die Ruhesuche angeschaltet werden muss. Dementsprechend muss beim eigentlichen Updatevorgang des Evaluationsvektors jeder Stellung am Ende einer Hauptvariante noch zusätzlich die Ruhesuche zur genauen Bewertung der Stellung mit veränderten Parametern verwendet werden.

5. Nach jeder Partie muss die Gültigkeit des Partieverlaufes überprüft und ein Resultat vergeben werden.

Problematisch bei UCI ist die Verwertung der Ergebnisse. Es gibt keine Notwendigkeit der Engine mitzuteilen: “du hast gewonnen”. Diese Verwaltung müssen wir selber übernehmen. Um zu entscheiden, ob eine Partie ein gültiges Resultat geliefert hat, gehen wir folgende Szenarien einmal durch:

1.Fall: Gegner wird matt gesetzt

2.Fall: Gegner gibt auf

3.Fall: Spiel endet Remis

4.Fall: Engine wird matt gesetzt

5.Fall: Engine verliert nach Zeit

Der 1., 3. und 4. Fall ist mit dem Schachmotor schnell zu überprüfen. Der 5.Fall tritt ein, wenn am Ende wenig Zeit zur Verfügung stand. Alle anderen Zustände werden Remis gewertet, falls eine Mindestzuganzahl von 20 Zügen[2] erfolgt ist (darunterliegende Zuganzahlen könnten z.B. auf einen Verbindungsabbruch zurückzuführen sein).

6. Falls ein positives Resultat erfolgt ist, wird die Partie abgespeichert.

Mit positivem Resultat meine ich hier einen Sieg oder eine Niederlage. Remispartien nehme ich für die Verwendung von TD-Leaf(λ) nicht. Nun müssen alle Partieergebnisse in eine weitere Datei gespeichert werden. Als Daten kommen die Zugfolge, Hauptvarianten, Zugbewertungen und der verwendete Vektor in Frage. Sollte die letzte Partie kein positives Resultat gebracht haben, so erstellen wir eine leere Partie mit gleichem Namen. Hinsichtlich der anschließenden Verwertung der Daten, wie sie für die Auswertung wichtig sind, habe ich noch ein zusätzlichen Mechanismus eingeführt. Jede Partie erhält einen Counter, der

[2] Das ist ein von mir willkürlich gesetzter Wert, um zu gewährleisten, dass es sich um “saubere” Schachpartie handelt.

fortlaufend die Partien mit ensprechenden Namen versieht. Nachdem eine Partie, zum updaten des Vektors verwendet wurde, kopiere ich sie in einen weiteren Ordner und dokumentiere dadurch die “evolutionäre Entwicklung” meines Bewertungsvektors. Der Dateiname der letzten Partie kann somit immer der gleiche sein. Das vereinfacht die Verwaltung erheblich. Dazu habe ich die Klasse *RLFileManager.cs* geschrieben. Ihre alleinige Aufgabe ist es, die Dateien entsprechend zu verwalten und eine Partiehistorie für spätere Auswertungen anzulegen.

7. Vor jeder Partie müssen die Parameter eingelesen und falls ein positives Resultat in der Vorgängerpartie vorhanden war, mit TD-Leaf(λ) angeglichen werden.

Der TD-Leaf-Algorithmus wie er bereits detailiert besprochen wurde, kann mit den entsprechenden Daten “gefüttert” und zur Berechnung des update-Vektors Verwendung finden.

8. Vorab ist eine Überlegung zu treffen, inwieweit die Parameter anzupassen sind. Es können beispielsweise die Materialfaktoren vorgegeben werden. Ebenfalls ist die Länge des Lernvorganges zu berücksichtigen.

Nun stellt sich noch die Frage, ob der Schachmotor beim Lernen auf bereits vorhandene Werte zurückgreifen darf oder nicht. Eine bereits recht gut eingestellte Evaluation kann oft noch ein wenig verbessert werden. Damit verringert sich die benötigte Lernzeit.

6.2 Visual Eval 1.0

Um den Lernvorgang besser zu verstehen und zu überprüfen, wurde die C#-Klasse `VisualEval.cs` implementiert. Damit lassen sich alle Evaluationsparameter übersichtlich anzeigen. Auch die Veränderungen der Bewertungsmatrizen können über einen längeren Zeitraum, z.B. die der letzten 100 Spiele verglichen und in einer Schleife vorgeführt werden. Siehe dazu Abbildung 6.1. Die grosse Anzahl von Evaluationsfaktoren machte es notwendig eine Farbdarstellung der Parameter zu wählen. Diese Farbwerte richten sich nach dem kleinsten und grössten Wert der Daten. Ausgehend von diesen wurde ein Farbintervall gewählt und alle dazwischenliegenden Farbwerte interpoliert.

6.3 Aktuelle Schachregeln nach der FIDE

Die aktuellen Schachregeln der FIDE sind unter folgendem Link zu finden:

http://www.schachbund.ch/federation/reglementations/index_d.html

6.4 Spielregeln von Tic-Tac-Toe

Tic-Tac-Toe ist ein Spiel für zwei Spieler. Als Basis dient eine leere 3×3-Matrix. Spieler 1 und Spieler 2 besitzen ein unterschiedliches Zeichen (meistens X und O). Gespielt wird abwechselnd. Spieler 1 fängt an sein Zeichen in eines der leeren Matrixeinträge zu schreiben, anschließend Spieler 2, usw. Ziel des Spieles ist es auf einer Linie (von oben nach unten oder von links nach rechts) oder Diagonalen 3 eigene Symbole einzutragen. Sollte ein solcher Zustand auftreten, bricht das Spiel sofort ab, der Gewinner steht fest.

6.5 Universal Chess Interface (UCI)

Um die Konzentration bei der Schachmotorentwicklung auf die wesentlichen Dinge zu konzentrieren, bieten sich zwei Protokolle an, mit denen Schachmotoren unter verschiedensten Oberflächen getestet und untersucht werden können.

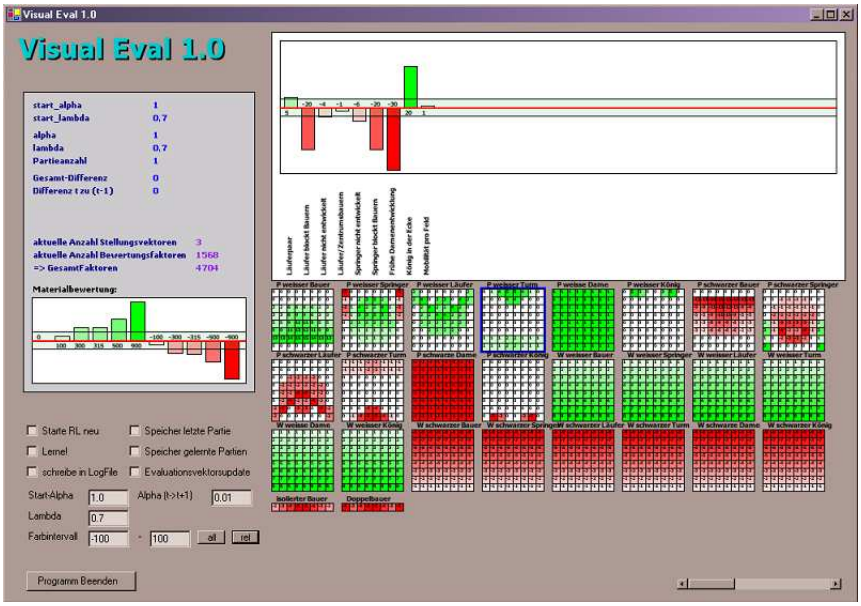

Abbildung 6.1: Visual Eval 1.0

Winboard war der erste Protokolltyp und wird auch immer noch verwendet.

http://www.tim-mann.org/xboard.html

Die Spezifikation des aktuelleren Protokolltyps UCI, den auch der Schachmotor *FUSc#* verwendet, ist unter folgendem link zu finden:

http://www.shredderchess.com/download.html

Zu erwähnen ist noch, dass Konvertierungsprogramme exisiteren, mit denen z.B. ein Winboard-Motor auch unter einer UCI-Oberfläche eingesetzt werden kann.

6.6 FUSc# auf www.schach.de

Unter dem Pseudonym *deepfusch* ist die aktuellste *FUSc#*-Version (im Maschinenraum) zu finden. Siehe Abbildung 6.2.

Für Amateurschachprogramme ist dieser Spielserver nicht zu empfehlen, wenn das Programm im Maschinenraum gegen andere Schachprogramme antreten soll, da dort fast ausschließlich die besten professionellen Programme gegeneinander spielen. Zeitweilig kommt es vor, dass über 50 Versionen von *Fritz* 8 spielen, einem der stärksten Schachprogramme. Das macht eigentlich nur Sinn, wenn die jeweiligen Fritz-Eröffnungsbücher aktualisiert werden sollen. Ein schwächerer Schachmotor hat dort kaum Chancen, Auskunft über seine wirkliche Performance zu erhalten.

Durch den Kontakt mit Thomas Mayer, der den Schachmotor *Quark* entwickelt und damit sogar den 9-ten Platz bei den Weltmeisterschaften 2001 in Paderborn belegt hat, gab es die Möglichkeit *FUSc#* eventuell doch im großen Spielsaal (normalerweise sind nur menschliche Spieler erlaubt) antreten zu lassen. Innerhalb von wenigen Stunden wurde das von Mathias Feist, Mitentwickler von Chessbase, ermöglicht. *FUSc#* spielt nun unter dem Namen *Comp Fusch* auf *http://www.schach.de*. Da in diesem Spielsaal täglich mehrere tausend Schachspieler aus aller Welt verkehren, besteht eine gute Möglichkeit, die Koeffizienten im Laufe der kommenden Monate gut zu trainieren.

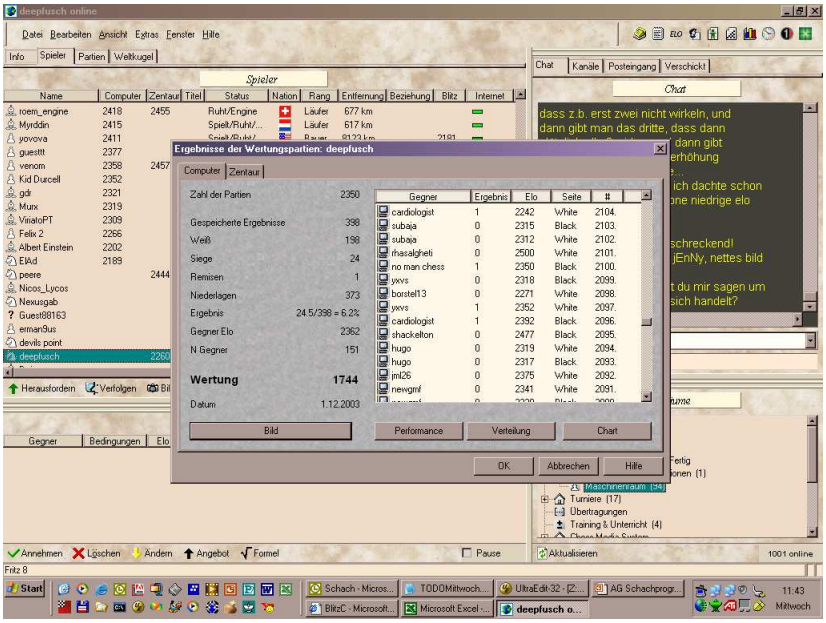

Abbildung 6.2: *FUSc#* spielt auf *www.schach.de*

Kleine Partiensammlung vom Schachserver

Comp Fusch - DerCloud7(1823) [29.07.2004] 1-0
3 Minuten-Blitzpartie (mit 2 Sekunden inkrement)

1.d4 Sh6 **2.**e4 Sg8 **3.**Sf3 d6 **4.**Ld3 g6 **5.**Sc3 Lg7 **6.**0-0 e6 **7.**Lf4 Se7 **8.**Lb5+ Ld7 **9.**Lc4 0-0 **10.**e5 d5 **11.**Ld3 Sbc6 **12.**Tc1 a6 **13.**Te1 Sf5 **14.**Lg5 Sfe7 **15.**a3 h6 **16.**Lf4 Kh7 **17.**h3 Sb8 **18.**b4 c6 **19.**g4 a5 **20.**bxa5 Dxa5 **21.**Sb1 b5 **22.**c3 Sc8 **23.**Te2 Sb6 **24.**h4 Sa4 **25.**h5 Le8 **26.**Te1 Sb2 **27.**hxg6+ fxg6 **28.**Lxg6+ Lxg6 **29.**Dd2 Sc4 **30.**Dd1 Sxa3 **31.**Sxa3 Dxa3 **32.**Ld2 Sd7 **33.**Sh4 c5 **34.**Sxg6 Kxg6 **35.**Tb1 Tab8 **36.**Ta1 Db2 **37.**Ta7 Sb6 **38.**f4 Sc4 **39.**Lc1 Dxc3 **40.**dxc5 Dd4+ **41.**Dxd4 DerCloud7 gibt auf

Comp Fusch - Ujima24(1583) [04.08.2004] 1-0
3 Minuten-Blitzpartie

1.d4 e5 **2.**dxe5 d6 **3.**Sf3 Sc6 **4.**exd6 Lxd6 **5.**Sbd2 Lg4 **6.**Se4 Lxf3 **7.**Sxd6+ Dxd6 **8.**Dxd6 cxd6 **9.**exf3 Sd4 **10.**Ld3 Sf6 **11.**0-0 0-0 **12.**Lf4 d5 **13.**c3 Sc6 **14.**Tfe1 Sh5 **15.**Ld6 Tfe8 **16.**a4 Te6 **17.**Txe6 fxe6 **18.**Te1 Te8 **19.**Lf5 Sd8 **20.**h4 Sf6 **21.**Ld3 Kf7 **22.**b4 Sc6 **23.**b5 Sd8 **24.**g4 b6 **25.**g5 Sg8 **26.**Lxh7 Sb7 **27.**g6+ Kf6 **28.**Le5+ Kf5 **29.**Lxg7 Sc5 **30.**Kg2 Sxa4 **31.**Ta1 Sxc3 **32.**Lxc3 Ujima24 gibt auf

Schachblocki(1823) - **Comp Fusch** [04.08.2004] 1-0
2 Minuten-Blitzpartie

1.Sf3 Sf6 **2.**g3 Sc6 **3.**Lg2 d5 **4.**c3 Lf5 **5.**d3 e5 **6.**Sbd2 a5 **7.**0-0 Le7 **8.**Te1 0-0 **9.**Dc2 e4 **10.**Sh4 exd3 **11.**exd3 Lg4 **12.**f3 Le6 **13.**Sb3 a4 **14.**Sd2 g5 **15.**Txe6 fxe6 **16.**Sg6 hxg6 **17.**d4 Kf7 **18.**Sf1 g4 **19.**fxg4 Sxg4 **20.**Lf4 e5 **21.**dxe5 Lc5+ **22.**Kh1 Sgxe5 **23.**Lxe5 Sxe5 **24.**Sd2 Kg8 **25.**Te1 Tf5 **26.**b4 Lf2 **27.**Te2 a3 **28.**Sb3 Lb6 **29.**Sd4 Lxd4 **30.**cxd4 Sc6 **31.**Dc5 Ta4 **32.**Lh3 Tg5 **33.**Le6+ Kg7 **34.**Tf2 Txb4 **35.**Tf7+ Kh6 **36.**Dc1 Dd6 **37.**h4 Dxg3 **38.**hxg5+ Dxg5 **39.**Th7+ Kxh7 **40.**Dxg5 Txd4 **41.**Lf7 Td1+ **42.**Kh2 Sb4 **43.**Dxg6+ Kh8 **44.**Dg8#

Comp Fusch - Akkaios(2215) [05.08.2004] 1-0
1 Minuten-Blitzpartie

1.e4 c5 **2.**Sf3 e6 **3.**d4 d6 **4.**dxc5 dxc5 **5.**Dxd8+ Kxd8 **6.**Sc3 a6 **7.**Lf4 Ke8 **8.**Td1 Sc6 **9.**h3 f6 **10.**Lc4 Sge7 **11.**a3 e5 **12.**Le3 Sd4 **13.**Kf1 b6 **14.**Se1 Lb7 **15.**Sa4 Sc8 **16.**c3 Sc6 **17.**La2 Le7 **18.**Ke2 Sa5 **19.**b4 cxb4 **20.**axb4 b5 **21.**bxa5 bxa4 **22.**f3 Sa7 **23.**Sc2 Sc6 **24.**Lb6 Ld8 **25.**h4 Lxb6 **26.**axb6 Td8 **27.**Txd8+ Kxd8 **28.**Lc4 Ke7 **29.**Ta1 Td8 **30.**Txa4 Sb8 **31.**Ta1 Td6 **32.**Tb1 Kd7 **33.**Se3 Kc6 **34.**Ld5+ Kd7 **35.**Lxb7 Ke7 **36.**Sf5+ Ke6 **37.**Lc8+ Akkoios verliert auf Zeit

Comp Fusch - Akkaios(2183) [05.08.2004] 1-0
1 Minuten-Blitzpartie

1.e4 c5 **2.**Sf3 e6 **3.**d4 cxd4 **4.**Sxd4 d6 **5.**Sc3 a6 **6.**Lc4 Sf6 **7.**Lb3 Dc7 **8.**0-0 b5 **9.**a4 b4 **10.**Sa2 a5 **11.**Le3 Sxe4 **12.**f3 Sf6 **13.**c3 d5 **14.**cxb4 axb4 **15.**Ld2 Db7 **16.**Tc1 Ld7 **17.**g4 h6 **18.**h3 Ld6 **19.**f4 0-0 **20.**f5 e5 **21.**Sf3 e4 **22.**Sd4 Sa6 **23.**a5 Tac8 **24.**h4 Txc1 **25.**Sxc1 Lc5 **26.**Le3 Da7 **27.**g5 hxg5 **28.**hxg5 Sg4 **29.**Dxg4 Lxd4 **30.**Lxd4 Dxd4+ **31.**Tf2 De5 **32.**Dd1 Tc8 **33.**g6 fxg6 **34.**Lxd5+ Kf8 **35.**fxg6+ Ke8 **36.**Sb3 Lf5 **37.**Lb7 Td8 **38.**Dh5 Dg3+ **39.**Tg2 De1+ **40.**Kh2 Df1 **41.**Lxa6 Td3 **42.**Lxd3 exd3 **43.**Dh4 Comp Fusch,DarkFUSC# V0.01 DEBU bietet Remis **43.**..d2 **44.**Dh8+ Ke7 **45.**Dxg7+ Kd6 **46.**Txd2+ Kc6 **47.**Sd4+ Kc5 **48.**De5+ Kc4 **49.**Db5#

Father(2051) - **Comp Fusch** [06.08.2004] 1/2-1/2
3 Minuten-Blitzpartie

1.d4 d5 **2.**e3 Sf6 **3.**c3 Lf5 **4.**Ld3 Lxd3 **5.**Dxd3 Sbd7 **6.**f4 c5 **7.**Sf3 cxd4 **8.**exd4 e6 **9.**0-0 Ld6 **10.**Te1 0-0 **11.**g3 Tc8 **12.**Sbd2 h6 **13.**Sf1 Se4 **14.**S3d2 Sdf6 **15.**Sxe4 Sxe4 **16.**Sd2 Sf6 **17.**Sf3 Se4 **18.**Sd2 Sf6 **19.**Sf3 a6 **20.**Ld2 b5 **21.**a3 Se4 **22.**Teb1 Sxd2 **23.**Dxd2 Df6 **24.**Kg2 Dd8 **25.**Kg1 Df6 **26.**Kg2 Dd8 **27.**Kg1 Remis

Son Goten(2311) - **Comp Fusch** [06.08.2004] 1/2-1/2
0 Minuten-Blitzpartie (mit 1 Sekunde inkrement)

1.d4 d5 **2.**e3 Sf6 **3.**Sf3 c5 **4.**Sbd2 cxd4 **5.**exd4 Lf5 **6.**c3 Sc6 **7.**Le2 h5 **8.**0-0 e6 **9.**Te1 Ld6 **10.**b3 0-0 **11.**Lb2 Tc8 **12.**Tc1 a6 **13.**c4 Te8 **14.**c5 Lf4 **15.**a3 b5 **16.**b4 Tf8 **17.**Db3 h4 **18.**a4 bxa4 **19.**Dxa4 Se4 **20.**Sxe4 dxe4 **21.**Se5 Sxd4 **22.**Lxd4 Lxc1 **23.**La1 Ld2 **24.**Td1 Dg5 **25.**Dxa6 Lxb4 **26.**c6 f6 **27.**Sd7 Tfd8 **28.**c7 Txc7 **29.**Sxf6+ Kf7 **30.**Txd8 Tc1+ **31.**Lf1 gxf6 **32.**Db7+ Le7 **33.**Td7 Txf1+ **34.**Kxf1 Dc1+ **35.**Ke2 Lg4+ **36.**f3 exf3+ **37.**gxf3 Lxf3+ **38.**Kxf3 Df1+ **39.**Kg4 Dg1+ **40.**Kxh4 Dxh2+ **41.**Kg4 f5+ **42.**Kf3 Dh1+ **43.**Ke3 Dg1+ **44.**Kd2 Dh2+ **45.**Kc3 De5+ **46.**Kc4 Dc5+ **47.**Kb3 Da3+ **48.**Kc2 Da4+ **49.**Kc1 Dxa1+ **50.**Kc2 Da4+ **51.**Kb2 Da3+ Son Goten bietet Remis **52.**Kc2 Dc5+ **53.**Kb1 Da3 Remis

ni.ke(1617) - **Comp Fusch** [06.08.2004] 0-1
3 Minuten-Blitzpartie

1.f4 c5 **2.**Sf3 Sc6 **3.**e3 g6 **4.**Le2 Lg7 **5.**0-0 d6 **6.**c4 Sf6 **7.**Dc2 e5 **8.**fxe5 dxe5 **9.**Td1 0-0 **10.**d4 cxd4 **11.**exd4 Sxd4 **12.**Sxd4 exd4 **13.**Ld3 Te8 **14.**Sd2 Sg4 **15.**Sf3 Se3 **16.**Lxe3 Txe3 **17.**Te1 Ld7 **18.**Txe3 dxe3 **19.**Te1 Lh6 **20.**g4 Lxg4 **21.**Le2 Lf5 **22.**Db3 b6 **23.**Tf1 a5 **24.**c5 bxc5 **25.**Lc4 a4 **26.**Dc3 Ta7 **27.**Kh1 Te7 **28.**Se5 Da8+ **29.**Sf3 e2 ni.ke. gibt auf

Son Goten(2314) - **Comp Fusch** [09.08.2004] 0-1
1 Minuten-Blitzpartie

1.d4 d5 **2.**e3 Sf6 **3.**Sf3 c5 **4.**Sbd2 cxd4 **5.**exd4 Lf5 **6.**c3 Sc6 **7.**Le2 h5 **8.**0-0 e6 **9.**Te1 Ld6 **10.**Sf1 0-0 **11.**Lg5 Lh7 **12.**Se3 g6 **13.**h4 a6 **14.**Ld3 b5 **15.**Lc2 Tb8 **16.**Dd3 Tb7 **17.**a3 Td7 **18.**g4 hxg4 **19.**Sxg4 Le7 **20.**Sge5 Sxe5 **21.**Sxe5 Tc7 **22.**h5 gxh5 **23.**Dd2 Lxc2 **24.**Dxc2 Se4 **25.**Txe4 dxe4 **26.**Lh6 Dxd4 Son Goten gibt auf

Kapitel 7

Literaturverzeichnis

zitierte Literatur:

[Awerbach] Awerbach, J., Kotow, A., Judowitsch, M.: *Schachbuch für Meister von Morgen*, Beyer Verlag 2003

[BaxTriWea] Baxter, Jonathan; Triddgell, Andrew; Weaver, Lex: *KnightCap:A chess program that learns by combining TD(lambda) with minimax search*, http://citeseer.ist.psu.edu/baxter98knightcap.html

[BaxTriWea2] Baxter, Jonathan; Triddgell, Andrew; Weaver, Lex: *Experiments in Parameter Learning Using Temporal Differences*, http://cs.anu.edu.au/~Lex.Weaver/pub_sem/publications/ICCA-98_equiv.pdf

[Beal] Beal, D.F.; Smith M.C.: *Learning piece values using temporal differences*, ICCA Journal, vol. 20, no. 3, pp. 147-151, 1997

[Block] Studienarbeit Block, Marco: *Reinforcement Learning in der Schachprogrammierung*

[First] Internetseite: *http://www.chez.com/cazaux/first-persian-russian.htm*

[Fürn2] Fürnkranz, Johannes; Kubat, Miroslav: *Machines That Learn To Play Games*, Nova Science Publisher 2001

[Gherrity] Gherrity, Michael: *A Game-Learning Machine*, Dissertation, University of California, San Diego, 1993.

[Heinz] Heinz, Ernst A.: *Scalable Search in Computer Chess*, Dissertation, Universität Karlsruhe, Vieweg 2000

[King] King, Daniel: *KASPAROW gegen DEEP BLUE*, Beyer Verlag 1997

[Luger] Luger, George F.: *Künstliche Intelligenz*

[Plaat] Plaat, Aske: *RESEARCH, RE:SEARCH & RE-SEARCH*, Tinbergen Institute, 1996, http://www.cs.vu.nl/~aske/Papers/abstr-ks.html

[Robertie] Robertie, Bill *http://www.gammonvillage.com-images-photos-Robertiea.jpg*

[Rojas] Rojas, R.: *Theorie der neuronalen Netze*, Springer-Verlag, 1993

[Samuel] Samuel, A.L.: *Some Studies in Machine Learning Using the Game of Checkers*. IBM Journal of Research and Development, 3:210-229, 1959

[Stein] Steinwender, D.; Friedel, F.: *Schach am PC*. Markt und Technik 1995

[Sutton] Sutton, Richard S.: *Learning to Predict by the Methods of Temporal Differences*. Machine Learning, 3:9-44, 1988

[Tesauro2] Tesauro, Gerald: *Comparison Training of Chess Evaluation Functions*, Kapitel 6 in [Fürn2]

[Tesauro3] Tesauro, Gerald: *Temporal Difference Learning and TD-Gammon*, Communications of the ACM, März 1995 / Vol. 38, No. 3

[Thrun] Thrun, Sebastian: *Learning To Play the Game of Chess*, http://www.cs.cmu.edu/~thrun/papers/thrun.nips7.neuro-chess.ps.gz

[Thong] Thong B. Trinh, Anwer S. Bashi, Nikhil Deshpande: *Temporal Difference Learning in Chinese Chess*. IEA/AIE (Vol. 2) 1998: 612-618

[Tridgell] Tridgell, Andrew: *KnightCap - a parallel chess program on the AP1000+*

[Zhang] Zhang, Byoung-Tak: *Lernen durch Genetisch-Neuronale Evolution: Aktive Anpassung an unbekannte Umgebungen mit selbstentwicklenden parallelen Netzwerken.* Infix 1992

[Zipproth] Zipproth, Stefan: *Suchet, so werdet ihr finden*. Artikel S.15 CSS(ComputerSchach & Spiele) 3/2003

weitere Literatur:

Baxter, Jonathan; Tridgell, Andrew; Weaver, Lex: *TDLeaf(lambda):Combining Temporal Difference Learning with Game-Tree Search.*

Baxter, Jonathan; Triddgell, Andrew; Weaver, Lex: *KnightCap:A chess program that learns by combining TD(lambda) with game-tree search*

Baxter, Jonathan; Triddgell, Andrew; Weaver, Lex: *Learning To Play Chess Using Temporal Differences*

Baxter, Jonathan; Weaver, Lex; Bartlett, Peter: *Direct Gradient-Based Reinforcement Learning: II. Gradient Ascent Algorithms and Experiments*

Fürnkranz, Johannes: *Machine Learning In Computer Chess: The Next Generation*

Harmon, Mance E.; Harmon, Stephanie S.: *Reinforcement Learning: A Tutorial*

Kendall, Graham; Whitwell, Glenn: *An Evolutionary Approach for the Tuning of a Chess Evaluation Function using Population Dynamics*

Lämmel; Cleve: *Künstliche Intelligenz*

Levinson, Robert; Weber, Ryan: *Chess Neighborhoods, Function Combination, and Reinforcement Learning*

Schaeffer, Jonathan; Hlynka, Markian; Jussila, Vili: *Temporal Difference Learning Applied to a High-Performance Game-Playing Program*

Schraudolph, Nicol N.; Dayan, Peter; Sejnowski, Terrence J.: *Temporal Difference Learning of Position Evaluation in the Game of Go*

Simon, Herbert A.: *The Game of Chess*

Sutton, Richard S.; Barto, Andrew G.: *Reinforcement Learning An Introduction*

Swafford II, James F.: *Experiments With TDLeaf(lambda)*

Swafford II, James F.: *Optimizing Parameter Learning using Temporal Differences*

Tao, Nigel; Baxter, Jonathan; Weaver, Lex: *A Multi-Agent, Policy-Gradient approach to Network Routing*

Tesauro, Gerald: *Practical Issues in Temporal Difference Learning*

Trinh, Thong B.; Bashi, Anwer S.; Deshpande, Nikhil: *Temporal Difference Learning in Chinese Chess*

Weaver, Lex; Baxter, Jonathan: STD(lambda): *learning state differences with TD(lambda)*

Weaver, Lex; Baxter, Jonathan: *Learning From State Differences: STD(lambda)*

Weaver, Lex; Baxter, Jonathan: *Reinforcement Learning From State and Temporal Differences*

Weaver, Lex; Tao, Nigel: *The Optimal Reward Baseline for Gradient-Based Reinforcement Learning*

Wellner, Jörg: *Reinforcement Learning Tutorial*, Vorlesung, SS 2000